بايثون للمبتدئين

المقدمة

مرحبًا بك في عالم بايثون المثير! إذا كنت جديدًا في عالم البرمجة أو تبحث عن لغة برمجة سهلة التعلم وقوية في نفس الوقت، فإن بايثون هي الخيار الأمثل لك. لقد أصبحت بايثون واحدة من أكثر لغات البرمجة شعبية في العالم، وذلك بفضل بساطتها ومرونتها ومجتمعها الكبير والنشط.

تُستخدم بايثون في مجموعة واسعة من المجالات، بدءًا من تطوير الويب وتطبيقات سطح المكتب، مرورًا بتحليل البيانات والذكاء الاصطناعي، وصولًا إلى الأتمتة والألعاب. هذه اللغة متعددة الاستخدامات وتوفر أدوات ومكتبات قوية تساعدك على إنجاز مهام معقدة بسهولة.

يهدف هذا الكتاب إلى أن يكون دليلك الشامل لتعلم أساسيات بايثون من الصفر. سنبدأ من المفاهيم الأساسية وننتقل تدريجيًا إلى مواضيع أكثر تقدمًا، مع التركيز على الأمثلة العملية والتمارين التطبيقية لتعزيز فهمك. بحلول نهاية هذا الكتاب، ستكون لديك قاعدة صلبة في بايثون تمكنك من بناء مشاريعك الخاصة واستكشاف مجالات أعمق في عالم البرمجة.

ما هي بايثون؟

بايثون هي لغة برمجة عالية المستوى، تفسيرية، تفاعلية، وموجهة للكائنات. تم إنشاؤها بواسطة جيدو فان روسم (Guido van Rossum) وتم إصدارها لأول مرة في عام 1991. تتميز بايثون ببنية جملية واضحة ومقروءة، مما يجعلها سهلة التعلم والاستخدام، حتى للمبتدئين.

الخصائص الرئيسية لبايثون:

- **سهلة التعلم والاستخدام:** تتميز ببنية جملية بسيطة تشبه اللغة الإنجليزية، مما يقلل من منحنى التعلم.
- **لغة تفسيرية:** لا تحتاج إلى تجميع (compilation) الكود قبل تشغيله، مما يسرع عملية التطوير.
- **متعددة المنصات:** تعمل على أنظمة تشغيل مختلفة مثل Windows، macOS، و Linux.
- **موجهة للكائنات (Object-Oriented):** تدعم البرمجة الكائنية التوجه، مما يساعد على تنظيم الكود وإعادة استخدامه.
- **لغة ديناميكية:** لا تحتاج إلى تحديد أنواع المتغيرات بشكل صريح.

- **مكتبات ضخمة:** تحتوي على مجموعة هائلة من المكتبات الجاهزة التي تغطي تقريبًا كل مجال يمكن تخيله، مما يوفر الوقت والجهد على المطورين.

لماذا نتعلم بايثون؟

هناك العديد من الأسباب التي تجعل بايثون خيارًا ممتازًا للتعلم، خاصة للمبتدئين:

1. **سهولة التعلم:** كما ذكرنا، بايثون مصممة لتكون سهلة القراءة والكتابة، مما يجعلها نقطة انطلاق رائعة للمبرمجين الجدد.
2. **الطلب المتزايد في سوق العمل:** بايثون هي واحدة من أكثر اللغات طلبًا في صناعة التكنولوجيا، مما يفتح لك العديد من الفرص الوظيفية في مجالات مثل تطوير الويب، علم البيانات، الذكاء الاصطناعي، والأمن السيبراني.
3. **تعدد الاستخدامات:** يمكنك استخدام بايثون لبناء أي شيء تقريبًا، من تطبيقات الويب البسيطة إلى أنظمة الذكاء الاصطناعي المعقدة.
4. **مجتمع كبير وداعم:** يوجد مجتمع ضخم من مطوري بايثون حول العالم، مما يعني أنك ستجد دائمًا المساعدة والموارد عندما تحتاج إليها.
5. **الإنتاجية العالية:** بفضل بساطتها ومكتباتها الغنية، يمكن للمطورين إنجاز المزيد من العمل في وقت أقل باستخدام بايثون.

كيفية إعداد بيئة العمل

قبل أن نبدأ في كتابة الأكواد، نحتاج إلى إعداد بيئة العمل الخاصة بنا. يتضمن ذلك تثبيت بايثون واختيار محرر أكواد مناسب.

1. تثبيت بايثون

أفضل طريقة لتثبيت بايثون هي من الموقع الرسمي. اتبع الخطوات التالية:

1. **زيارة الموقع الرسمي:** اذهب إلى python.org/downloads.
2. **تنزيل المثبت:** سيقوم الموقع تلقائيًا باكتشاف نظام التشغيل الخاص بك واقتراح أحدث إصدار مستقر من بايثون. انقر على زر التنزيل.
3. **تشغيل المثبت:** بعد اكتمال التنزيل، قم بتشغيل الملف الذي تم تنزيله.
    - **ملاحظة هامة:** أثناء عملية التثبيت، تأكد من تحديد مربع الاختيار **"Add Python X.X to PATH"** (حيث X.X هو رقم الإصدار). هذا سيجعل بايثون متاحة من سطر الأوامر.
4. **إكمال التثبيت:** اتبع التعليمات على الشاشة لإكمال عملية التثبيت.

2. التحقق من التثبيت

للتأكد من أن بايثون قد تم تثبيتها بشكل صحيح، افتح موجه الأوامر (Command Prompt في Windows أو Terminal في macOS/Linux) واكتب الأمر التالي:

```
python --version
```

يجب أن ترى رقم إصدار بايثون الذي قمت بتثبيته (على سبيل المثال، `Python 3.9.7`).

3. اختيار محرر الأكواد

محرر الأكواد هو المكان الذي ستكتب فيه أكواد بايثون الخاصة بك. هناك العديد من الخيارات الممتازة المتاحة، وإليك بعضها:

- **Visual Studio Code (VS Code):** محرر أكواد مجاني وقوي من Microsoft، ويدعم العديد من اللغات والإضافات. إنه خيار ممتاز للمبتدئين والمحترفين على حد سواء.
- **PyCharm:** بيئة تطوير متكاملة (IDE) مخصصة لبايثون من JetBrains. توفر PyCharm ميزات قوية مثل التصحيح التلقائي، وتصحيح الأخطاء (debugging)، وإدارة المشاريع. هناك نسخة مجانية (Community Edition) مناسبة للمبتدئين.
- **IDLE:** يأتي مع تثبيت بايثون الافتراضي. إنه محرر بسيط ومناسب لتجربة الأكواد السريعة وتعلم الأساسيات.

نوصي باستخدام VS Code أو PyCharm للحصول على تجربة تطوير أفضل. يمكنك تنزيل VS Code من code.visualstudio.com و PyCharm من jetbrains.com/pycharm.

الآن بعد أن أصبحت بيئة العمل جاهزة، دعنا نبدأ رحلتنا في تعلم بايثون!

الفصل الأول: الأساسيات

في هذا الفصل، سنغوص في أساسيات لغة بايثون. سنتعلم كيفية التعامل مع المتغيرات، وأنواع البيانات المختلفة، وكيفية إجراء العمليات الحسابية والمنطقية، بالإضافة إلى كيفية إدخال البيانات وإخراجها. هذه المفاهيم هي اللبنات الأساسية لأي برنامج بايثون.

المتغيرات وأنواع البيانات

المتغيرات هي حاويات لتخزين البيانات. يمكنك التفكير فيها كصناديق تحمل قيمًا معينة. في بايثون، لا تحتاج إلى تحديد نوع المتغير عند إنشائه؛ بايثون تحدد النوع تلقائيًا بناءً على القيمة التي تسندها إليه.

قواعد تسمية المتغيرات:

- يجب أن تبدأ بحرف (a-z, A-Z) أو شرطة سفلية (_).
- لا يمكن أن تبدأ برقم.
- يمكن أن تحتوي على أحرف وأرقام وشرطات سفلية فقط.
- حساسة لحالة الأحرف (myVariable يختلف عن myvariable).
- لا يمكن أن تكون كلمة محجوزة في بايثون (مثل `while`, `for`, `if`).

مثال على تعريف المتغيرات:

```
# تعريف متغيرات من أنواع مختلفة
name = "أحمد"        # سلسلة نصية (string)
age = 30             # عدد صحيح (integer)
height = 1.75        # عدد عشري (float)
is_student = True    # قيمة منطقية (boolean)

print(name)
print(age)
print(height)
print(is_student)
```

أنواع البيانات الأساسية:

1. **الأعداد (Numbers):**

   - **الأعداد الصحيحة (`int` - Integers):** أعداد كاملة بدون كسور (مثل 10, -5, 1000).
   - **الأعداد العشرية (`float` - Floats):** أعداد تحتوي على جزء عشري (مثل 3.14, -0.5, 2.0).

`python num1 = 10 num2 = 3.14 print(type(num1)) # <class 'int'> print(type(num2)) # <class 'float'>`

2. **السلاسل النصية (`str` - Strings):** تسلسل من الأحرف محاط بعلامات اقتباس مفردة (`'`) أو مزدوجة (`"`).

`python greeting = "مرحبًا بالعالم!" city = 'الرياض' print(greeting) print(city) print(type(greeting)) # <class 'str'>`

3. **القيم المنطقية (`bool` - Booleans):** تمثل قيمتين فقط: `True` (صحيح) أو `False` (خطأ). تُستخدم في عمليات المقارنة والتحكم في سير البرنامج.

```
python is_active = True has_permission = False print(is_active)
<'print(has_permission) print(type(is_active)) # <class 'bool
```

4. **القوائم (`list` - Lists):** مجموعة مرتبة وقابلة للتغيير من العناصر. يمكن أن تحتوي على عناصر من أنواع بيانات مختلفة.

```
mixed_list = [1, "hello", ["موز" ,"برتقال" ,"تفاح"] = python fruits
3.14, True] print(fruits) print(mixed_list) print(type(fruits)) #
<'<class 'list
```

5. **القواميس (`dict` - Dictionaries):** مجموعة غير مرتبة وقابلة للتغيير من أزواج المفتاح-القيمة (key-value pairs). كل مفتاح يجب أن يكون فريدًا.

```
{"جدة" :"age": 25, "city" ,"علي" :"name"} = python person
print(person["name"]) print(person["age"]) print(type(person)) #
<'<class 'dict
```

6. **المجموعات (`set` - Sets):** مجموعة غير مرتبة وغير مفهرسة من العناصر الفريدة. لا تسمح بالعناصر المكررة.

```
python unique_numbers = {1, 2, 3, 2, 1} print(unique_numbers) # {1, 2,
<'3} print(type(unique_numbers)) # <class 'set
```

7. **الصفوف (`tuple` - Tuples):** مجموعة مرتبة وغير قابلة للتغيير من العناصر. تشبه القوائم ولكن لا يمكن تعديل عناصرها بعد إنشائها.

```
python coordinates = (10, 20) print(coordinates)
<'print(type(coordinates)) # <class 'tuple
```

العمليات الحسابية والمنطقية

العمليات الحسابية (Arithmetic Operators):

العملية	الوصف	مثال	النتيجة
`+`	الجمع	`3 + 5`	`8`
`-`	الطرح	`4 - 10`	`6`
`*`	الضرب	`6 * 2`	`12`
`/`	القسمة	`3 / 10`	`3.33`
`//`	القسمة الصحيحة	`3 // 10`	`3`
`%`	باقي القسمة	`3 % 10`	`1`
`**`	الأس	`3 ** 2`	`8`

أمثلة:

```
 x = 15
y = 4

print(x + y)  # 19
print(x - y)  # 11
print(x * y)  # 60
print(x / y)  # 3.75
print(x // y) # 3
print(x % y)  # 3
print(x ** 2) # 225
```

العمليات المنطقية (Comparison Operators):

تُستخدم لمقارنة قيمتين وتُرجع قيمة منطقية (`True` أو `False`).

| العملية | الوصف | مثال | النتيجة | | :----- | :--------------- | :------------ | :------ |\n| `==` | يساوي | `5 == 5` | `True` | | `=!` | لا يساوي | `3 =! 5` | `True` | | `<` | أكبر من | `5 < 10` | `True` | | `>` | أصغر من | `10 > 5` | `True` | | `=<` | أكبر من أو يساوي | `10 =< 10` | `True` | | `=>` | أصغر من أو يساوي | `8 => 5` | `True` |

أمثلة:

```
a = 10
b = 20

print(a == b) # False
print(a != b) # True
print(a > b)  # False
print(a < b)  # True
print(a >= 10) # True
print(b <= 20) # True
```

العمليات المنطقية (Logical Operators):

تُستخدم لدمج الشروط المنطقية.

العملية	الوصف	مثال	النتيجة
`and`	صحيح إذا كان كلا الشرطين صحيحين	`True and False`	`False`
`or`	صحيح إذا كان أحد الشرطين صحيحًا على الأقل	`True or False`	`True`
`not`	يعكس القيمة المنطقية	`not True`	`False`

أمثلة:

```
age = 25
has_license = True

print(age > 18 and has_license) # True (إذا كان العمر أكبر من 18 ولديه رخصة)
print(age < 18 or not has_license) # False (إذا كان العمر أصغر من 18 أو ليس لديه رخصة)
print(not has_license) # False
```

الإدخال والإخراج (Input/Output)

تُعد القدرة على إدخال البيانات من المستخدم وإخراج النتائج على الشاشة أمرًا أساسيًا في أي برنامج.

دالة `print()`:

تُستخدم دالة `print()` لإخراج البيانات إلى الشاشة (وحدة التحكم).

```python
print("مرحبًا بك في بايثون!")

name = "سارة"
print("اسم المستخدم:", name)

# طباعة عدة قيم مفصولة بمسافة افتراضية
print("العمر:", 30, "المدينة:", "القاهرة")

# تغيير الفاصل (sep) ونهاية السطر (end)
print("Hello", "World", sep="-") # Hello-World
print("Line 1", end=" ")
print("Line 2") # Line 1 Line 2
```

دالة `input()`:

تُستخدم دالة `input()` لأخذ مدخلات من المستخدم. تقوم بإرجاع المدخلات كسلسلة نصية (string)، حتى لو أدخل المستخدم أرقامًا.

```python
# طلب اسم المستخدم
user_name = input("أدخل اسمك: ")
print("أهلاً بك يا", user_name)

# طلب العمر وتحويله إلى عدد صحيح
age_str = input("أدخل عمرك: ")
age_int = int(age_str) # تحويل السلسلة النصية إلى عدد صحيح
print("عمرك هو:", age_int, "سنة.")

# طلب سعر منتج وتحويله إلى عدد عشري
price_str = input("أدخل سعر المنتج: ")
price_float = float(price_str) # تحويل السلسلة النصية إلى عدد عشري
print("سعر المنتج هو:", price_float, "ريال.")
```

ملاحظة هامة: تذكر دائمًا تحويل المدخلات من `input()` إلى النوع المناسب (مثل `int()` أو `float()`) إذا كنت تنوي إجراء عمليات حسابية عليها.

التعليقات (Comments)

التعليقات هي أجزاء من الكود يتجاهلها المفسر (interpreter) ولا يتم تنفيذها. تُستخدم التعليقات لشرح الكود، مما يجعله أكثر قابلية للقراءة والفهم للمبرمجين الآخرين أو لنفسك في المستقبل.

أنواع التعليقات:

1. **تعليق سطر واحد:** يبدأ بعلامة `#`.

python```

هذا تعليق سطر واحد

``` x = 10 # يمكن وضع التعليق بعد الكود في نفس السطر

2. **تعليق متعدد الأسطر (Docstrings):** يمكن استخدام ثلاث علامات اقتباس مفردة أو مزدوجة (`'''` أو `"""`) لإنشاء تعليقات تمتد على عدة أسطر. تُستخدم هذه عادةً لتوثيق الدوال والفئات والوحدات.

python``` """هذا تعليق يمتد على عدة أسطر. يمكن استخدامه لشرح أجزاء كبيرة من الكود. """

''' وهذا مثال آخر لتعليق متعدد الأسطر باستخدام علامات اقتباس مفردة. ''' ```

**أهمية التعليقات:**

- **شرح الكود:** توضح الغرض من أجزاء معينة من الكود.
- **تحسين القراءة:** تجعل الكود أسهل في الفهم والصيانة.
- **تصحيح الأخطاء:** يمكن استخدامها لتعطيل أجزاء من الكود مؤقتًا أثناء تصحيح الأخطاء.

## تمارين الفصل الأول

### التمرين 1:

أنشئ متغيرًا باسم `product_name` وأسند إليه قيمة "كمبيوتر محمول". أنشئ متغيرًا باسم `quantity` وأسند إليه قيمة `3`. أنشئ متغيرًا باسم `unit_price` وأسند إليه قيمة `1200.50`. احسب السعر الإجمالي (quantity * unit_price) وخزنه في متغير باسم `total_price`. اطبع `product_name` و `total_price` على الشاشة.

### التمرين 2:

اطلب من المستخدم إدخال اسمه وعمره. اطبع رسالة ترحيب تتضمن اسم المستخدم وعمره. تأكد من أن العمر يتم تخزينه كعدد صحيح.

### التمرين 3:

أنشئ متغيرين `num_a = 10` و `num_b = 4`. قم بإجراء العمليات الحسابية التالية واطبع النتائج: * الجمع * الطرح * الضرب * القسمة (العادية) * القسمة الصحيحة * باقي القسمة * الأس (num_a مرفوعًا للقوة num_b)
```

التمرين 4:

أنشئ متغيرًا `is_raining = True` ومتغيرًا `has_umbrella = False`. استخدم العمليات المنطقية `and` و `or` و `not` لطباعة النتائج التالية: * هل تمطر ولديه مظلة؟ * هل تمطر أو ليس لديه مظلة؟ * هل لا تمطر؟

حلول تمارين الفصل الأول

حل التمرين 1:

```
product_name = "كمبيوتر محمول"
quantity = 3
unit_price = 1200.50

total_price = quantity * unit_price

print("اسم المنتج:", product_name)
print("السعر الإجمالي:", total_price)
```

حل التمرين 2:

```
user_name = input("أدخل اسمك: ")
age_str = input("أدخل عمرك: ")
age_int = int(age_str)

print("أهلاً بك يا", user_name, "! عمرك هو", age_int, "سنة.")
```

حل التمرين 3:

```
num_a = 10
num_b = 4

print("الجمع:", num_a + num_b)
print("الطرح:", num_a - num_b)
print("الضرب:", num_a * num_b)
print("القسمة:", num_a / num_b)
print("القسمة الصحيحة:", num_a // num_b)
print("باقي القسمة:", num_a % num_b)
print("الأس:", num_a ** num_b)
```

حل التمرين 4:

```
is_raining = True
has_umbrella = False

print("هل تمطر ولديه مظلة؟", is_raining and has_umbrella)
print("هل تمطر أو ليس لديه مظلة؟", is_raining or not has_umbrella)
print("هل لا تمطر؟", not is_raining)
```

الفصل الثاني: هياكل التحكم

في هذا الفصل، سنتعلم كيفية التحكم في تدفق تنفيذ البرنامج باستخدام هياكل التحكم. هذه الهياكل تسمح لنا باتخاذ قرارات بناءً على شروط معينة، وتكرار تنفيذ كتل من التعليمات البرمجية عدة مرات.

الشروط (Conditional Statements)

تُستخدم الشروط لتنفيذ كود معين فقط إذا كان شرط معين صحيحًا. في بايثون، نستخدم `if`, `elif` (اختصار لـ else if), و `else`.

1. جملة `if`:

تُستخدم لتنفيذ كتلة من الكود إذا كان الشرط صحيحًا.

```
age = 18

if age >= 18:
    print("أنت مؤهل للتصويت.")
```

2. جملة `if-else`:

تُستخدم لتنفيذ كتلة من الكود إذا كان الشرط صحيحًا، وكتلة أخرى إذا كان الشرط خاطئًا.

```
score = 75

if score >= 50:
    print("لقد نجحت!")
else:
    print("لم تنجح.")
```

3. جملة `if-elif-else`:

تُستخدم للتحقق من عدة شروط بالتسلسل. يتم تنفيذ أول شرط يكون صحيحًا، وتُتجاهل باقي الشروط.

```
temperature = 25

if temperature > 30:
    print("الجو حار جداً.")
elif temperature > 20:
    print("الجو معتدل.")
else:
    print("الجو بارد.")
```

ملاحظات هامة:

- **المسافة البادئة (Indentation):** تعتمد بايثون على المسافات البادئة (المسافات البيضاء في بداية السطر) لتحديد كتل الكود. يجب أن تكون جميع الأسطر داخل نفس الكتلة بنفس المسافة البادئة (عادة 4 مسافات أو علامة تبويب واحدة).
- **النقطتان الرأسيتان (`:`):** يجب أن تتبع كل جملة `else`, `elif`, `if` بنقطتين رأسيتين.

الحلقات التكرارية (Loops)

تُستخدم الحلقات التكرارية لتنفيذ كتلة من الكود عدة مرات. بايثون توفر نوعين رئيسيين من الحلقات: `for` و `while`.

1. حلقة `for`:

تُستخدم للتكرار على تسلسل (مثل قائمة، سلسلة نصية، صف، أو مدى من الأرقام).

التكرار على قائمة:

```
fruits = ["تفاح", "برتقال", "موز"]
for fruit in fruits:
    print(fruit)
```

التكرار على سلسلة نصية:

```
word = "بايثون"
for char in word:
    print(char)
```

التكرار باستخدام `range()`:

تُستخدم دالة `range()` لإنشاء تسلسل من الأرقام. يمكن استخدامها بثلاث طرق:

- `range(stop)`: ينشئ تسلسلاً من 0 إلى `stop-1`.
- `range(start, stop)`: ينشئ تسلسلاً من `start` إلى `stop-1`.
- `range(start, stop, step)`: ينشئ تسلسلاً من `start` إلى `stop-1` بزيادة `step`.

```
# طباعة الأرقام من 0 إلى 4
for i in range(5):
    print(i)

# طباعة الأرقام من 2 إلى 5
for i in range(2, 6):
    print(i)

# طباعة الأرقام الزوجية من 0 إلى 10
for i in range(0, 11, 2):
    print(i)
```

2. حلقة `while`:

تُستخدم لتكرار كتلة من الكود طالما أن الشرط المحدد صحيح.

```
count = 0
while count < 5:
    print("العدد هو:", count)
    count += 1 # زيادة العدد بمقدار 1
```

ملاحظة هامة: تأكد دائمًا من وجود شرط ينهي حلقة `while` لتجنب الحلقات اللانهائية.

`break`, `continue`, `pass`

تُستخدم هذه الكلمات المفتاحية للتحكم بشكل أدق في سلوك الحلقات.

1. `break`:

تُستخدم لإنهاء الحلقة فورًا، حتى لو كان الشرط لا يزال صحيحًا أو لم يتم الانتهاء من التكرار على جميع العناصر.

```python
for i in range(10):
    if i == 5:
        break
    print(i)
# الناتج: 0, 1, 2, 3, 4
```

2. `continue`:

تُستخدم لتخطي التكرار الحالي والانتقال إلى التكرار التالي في الحلقة.

```python
for i in range(5):
    if i == 2:
        continue
    print(i)
# الناتج: 0, 1, 3, 4
```

3. `pass`:

هي جملة لا تفعل شيئًا. تُستخدم كعنصر نائب عندما تتطلب البنية جملة ولكنك لا تريد تنفيذ أي كود.

```python
# مثال على استخدام pass في جملة if
if True:
    pass # لا تفعل شيئًا
else:
    print("هذا لن يطبع")

# مثال على استخدام pass في دالة (سنشرح الدوال لاحقًا)
def my_function():
    pass # دالة فارغة حالياً
```

تمارين الفصل الثاني

التمرين 1:

اطلب من المستخدم إدخال رقم. إذا كان الرقم زوجيًا، اطبع "الرقم زوجي". وإذا كان فرديًا، اطبع "الرقم فردي".

التمرين 2:

اكتب برنامجًا يطبع الأرقام من 1 إلى 10 باستخدام حلقة `for`.

التمرين 3:

اكتب برنامجًا يطلب من المستخدم إدخال كلمة سر. استمر في طلب كلمة السر حتى يدخل المستخدم "secret". في كل مرة يدخل فيها كلمة سر خاطئة، اطبع "كلمة سر خاطئة، حاول مرة أخرى.". عندما يدخل كلمة السر الصحيحة، اطبع "مرحباً بك!".

التمرين 4:

اكتب برنامجًا يطبع الأرقام من 1 إلى 10، ولكن تخطى الرقم 5 باستخدام `continue`.

التمرين 5:

اكتب برنامجًا يطبع الأرقام من 1 إلى 10، ولكن توقف عند الرقم 7 باستخدام `break`.

حلول تمارين الفصل الثاني

حل التمرين 1:

```
num = int(input("أدخل رقماً: "))

if num % 2 == 0:
    print("الرقم زوجي")
else:
    print("الرقم فردي")
```

حل التمرين 2:

```
for i in range(1, 11):
    print(i)
```

حل التمرين 3:

```
password = ""
while password != "secret":
    password = input("أدخل كلمة السر: ")
    if password != "secret":
        print("كلمة سر خاطئة، حاول مرة أخرى.")
print("مرحباً بك!")
```

حل التمرين 4:

```python
for i in range(1, 11):
    if i == 5:
        continue
    print(i)
```

حل التمرين 5:

```python
for i in range(1, 11):
    if i == 7:
        break
    print(i)
```

الفصل الثالث: الدوال

الدوال (Functions) هي كتل من التعليمات البرمجية التي تؤدي مهمة محددة. تُستخدم الدوال لتنظيم الكود، وجعله أكثر قابلية للقراءة، وإعادة الاستخدام، وتجنب تكرار الكود. عندما تكتب دالة، يمكنك استدعاؤها عدة مرات من أجزاء مختلفة من برنامجك.

تعريف الدوال واستدعاؤها

لتعريف دالة في بايثون، نستخدم الكلمة المفتاحية `def` ، متبوعة باسم الدالة، ثم أقواس `()` ، ونقطتين رأسيتين `:` . الكود داخل الدالة يجب أن يكون بمسافة بادئة.

```python
# تعريف دالة بسيطة لا تأخذ أي معاملات ولا تُرجع قيمة
def greet():
    print("أهلاً بك في دالة بايثون!")

# استدعاء الدالة
greet()
```

المعاملات (Parameters) والقيم المعادة (Return Values)

يمكن للدوال أن تأخذ **معاملات** (inputs) وتُرجع **قيمًا** (outputs).

- **المعاملات:** هي المتغيرات التي تُمرر إلى الدالة عند استدعائها. تُعرف داخل أقواس الدالة.
- **القيم المعادة:** هي القيمة التي تُرجعها الدالة بعد إكمال مهمتها. نستخدم الكلمة المفتاحية `return` لإرجاع قيمة.

مثال على دالة تأخذ معاملات:

```python
def add_numbers(num1, num2):
    sum_result = num1 + num2
    print("مجموع الرقمين هو:", sum_result)

add_numbers(5, 3)  # استدعاء الدالة مع تمرير قيم
add_numbers(10, 20)
```

مثال على دالة تُرجع قيمة:

```python
def multiply_numbers(num1, num2):
    product = num1 * num2
    return product # إرجاع قيمة المنتج

result = multiply_numbers(4, 6) # تخزين القيمة المعادة في متغير
print("ناتج الضرب هو:", result)

print(multiply_numbers(7, 2)) # طباعة القيمة المعادة مباشرة
```

يمكن للدالة أن تُرجع أكثر من قيمة واحدة كـ `tuple`.

```python
def calculate_stats(numbers):
    total = sum(numbers)
    average = total / len(numbers)
    return total, average

my_list = [10, 20, 30, 40, 50]
sum_val, avg_val = calculate_stats(my_list)
print("المجموع:", sum_val, "المتوسط:", avg_val)
```

الدوال المدمجة (Built-in Functions)

بايثون تأتي مع مجموعة كبيرة من الدوال المدمجة التي يمكنك استخدامها مباشرة دون الحاجة إلى تعريفها. لقد استخدمنا بعضها بالفعل مثل `print()`, `input()`, `int()`, `float()`, `type()`, `len()`, `sum()`, `range()`.

بعض الدوال المدمجة الشائعة الأخرى:

- `abs()`: تُرجع القيمة المطلقة لعدد.
- `max()`: تُرجع أكبر عنصر في تسلسل أو أكبر قيمة بين عدة وسائط.
- `min()`: تُرجع أصغر عنصر في تسلسل أو أصغر قيمة بين عدة وسائط.
- `round()`: تُقرب عددًا إلى أقرب عدد صحيح أو إلى عدد محدد من المنازل العشرية.

- `str()`: تُحول قيمة إلى سلسلة نصية.

```
 print(abs(-10))     # 10
print(max(1, 5, 2)) # 5
print(min([10, 2, 8])) # 2
print(round(3.14159, 2)) # 3.14
print(str(123) + " هو رقم") # 123 هو رقم
```

نطاق المتغيرات (Variable Scope)

نطاق المتغير يشير إلى الجزء من البرنامج حيث يمكن الوصول إلى المتغير. في بايثون، هناك نوعان رئيسيان من النطاق:

1. **النطاق المحلي (Local Scope):** المتغيرات المعرفة داخل دالة تكون ذات نطاق محلي، مما يعني أنها لا يمكن الوصول إليها إلا من داخل تلك الدالة.

   ```python def my_function(): x = 10 # x هو متغير محلي print(x)

my_function()

# print(x) # سيؤدي هذا إلى خطأ لأن x غير معرف خارج الدالة

---

```
```
```

1. **النطاق العام (Global Scope):** المتغيرات المعرفة خارج أي دالة تكون ذات نطاق عام، ويمكن الوصول إليها من أي مكان في البرنامج، بما في ذلك داخل الدوال.

   ```python y = 20 # y هو متغير عام

def another_function(): print(y) # يمكن الوصول إلى y من داخل الدالة

another_function() print(y) ```

#### الكلمة المفتاحية `global`:

إذا كنت بحاجة إلى تعديل متغير عام من داخل دالة، يجب عليك استخدام الكلمة المفتاحية `global`.
   ```

```
counter = 0 # متغير عام

def increment_counter():
    global counter # الإعلان عن أننا نريد تعديل المتغير العام counter
    counter += 1
    print("العداد داخل الدالة:", counter)

print("العداد قبل الاستدعاء:", counter)
increment_counter()
increment_counter()
print("العداد بعد الاستدعاء:", counter)
```

تمارين الفصل الثالث

التمرين 1:

اكتب دالة باسم `calculate_area` تأخذ طول وعرض مستطيل كمعاملات، وتُرجع مساحة المستطيل. استدعِ الدالة مع قيمتين من اختيارك واطبع النتيجة.

التمرين 2:

اكتب دالة باسم `is_even` تأخذ عددًا صحيحًا كمعامل، وتُرجع `True` إذا كان العدد زوجيًا، و `False` إذا كان فرديًا. اختبر الدالة مع أرقام زوجية وفردية.

التمرين 3:

اكتب دالة باسم `get_full_name` تأخذ الاسم الأول والاسم الأخير كمعاملات، وتُرجع الاسم الكامل (الاسم الأول متبوعًا بمسافة ثم الاسم الأخير). استدعِ الدالة واطبع الاسم الكامل.

التمرين 4:

أنشئ متغيرًا عامًا باسم `balance` بقيمة `1000`. اكتب دالة باسم `withdraw` تأخذ مبلغًا كمعامل. يجب أن تقوم الدالة بطرح المبلغ من `balance` وتطبع الرصيد الجديد. تأكد من استخدام الكلمة المفتاحية `global` لتعديل `balance`. استدعِ الدالة `withdraw` مرتين بمبالغ مختلفة.

حلول تمارين الفصل الثالث

حل التمرين 1:

```python
def calculate_area(length, width):
    area = length * width
    return area

rectangle_area = calculate_area(10, 5)
print("مساحة المستطيل هي:", rectangle_area)
```

حل التمرين 2:

```python
def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False

print(is_even(4))  # True
print(is_even(7))  # False
```

حل التمرين 3:

```python
def get_full_name(first_name, last_name):
    full_name = first_name + " " + last_name
    return full_name

name = get_full_name("محمد", "أحمد")
print("الاسم الكامل:", name)
```

حل التمرين 4:

```python
balance = 1000 # متغير عام

def withdraw(amount):
    global balance
    if amount <= balance:
        balance -= amount
        print("تم سحب", amount, "الرصيد الجديد:", balance)
    else:
        print("رصيد غير كافٍ.")

print("الرصيد الأولي:", balance)
withdraw(200)
withdraw(500)
withdraw(400) # مثال على رصيد غير كافٍ
print("الرصيد النهائي:", balance)
```

الفصل الرابع: هياكل البيانات المتقدمة

في هذا الفصل، سنتعمق أكثر في هياكل البيانات الأساسية في بايثون: القوائم (Lists)، القواميس (Dictionaries)، والمجموعات (Sets). سنتعلم عمليات أكثر تقدمًا وكيفية استخدامها بفعالية في برامجنا.

القوائم (Lists) - عمليات متقدمة

القوائم هي هياكل بيانات مرنة وقوية. لقد رأينا بالفعل كيفية إنشائها والوصول إلى عناصرها. الآن، دعنا نستكشف بعض العمليات المتقدمة.

1. إضافة عناصر:

- `append(item)`: تُضيف عنصرًا واحدًا إلى نهاية القائمة.
- `extend(iterable)`: تُضيف عناصر من كائن قابل للتكرار (مثل قائمة أخرى) إلى نهاية القائمة.
- `insert(index, item)`: تُضيف عنصرًا في موقع محدد.

```
 my_list = [1, 2, 3]
my_list.append(4) # [1, 2, 3, 4]
print(my_list)

my_list.extend([5, 6]) # [1, 2, 3, 4, 5, 6]
print(my_list)

my_list.insert(0, 0) # [0, 1, 2, 3, 4, 5, 6]
print(my_list)
```

2. إزالة عناصر:

- `remove(item)`: تُزيل أول ظهور للعنصر المحدد.
- `pop(index)`: تُزيل العنصر في الموقع المحدد وتُرجعه. إذا لم يتم تحديد فهرس، تُزيل آخر عنصر.
- `clear()`: تُزيل جميع العناصر من القائمة.
- `del list[index]`: تُزيل العنصر في الموقع المحدد.
- `del list[start:end]`: تُزيل شريحة من القائمة.

```
my_list = [1, 2, 3, 4, 2]
my_list.remove(2) # [1, 3, 4, 2] - 2 تُزيل أول
print(my_list)

popped_item = my_list.pop() # popped_item = 2, my_list = [1, 3, 4]
print(popped_item, my_list)

del my_list[0] # [3, 4]
print(my_list)

my_list = [1, 2, 3, 4, 5]
del my_list[1:3] # [1, 4, 5] - 3 و 2 تُزيل
print(my_list)

my_list.clear() # []
print(my_list)
```

3. عمليات أخرى مفيدة:

- `index(item)` : تُرجع فهرس أول ظهور للعنصر المحدد.
- `count(item)` : تُرجع عدد مرات ظهور العنصر المحدد.
- `sort()` : تُرتّب القائمة في مكانها (تصاعديًا افتراضيًا).
- `sorted(list)` : تُرجع قائمة جديدة مرتبة دون تعديل القائمة الأصلية.
- `reverse()` : تعكس ترتيب عناصر القائمة في مكانها.
- `copy()` : تُرجع نسخة سطحية من القائمة.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
print(numbers.index(4)) # 2
print(numbers.count(1)) # 2

numbers.sort() # [1, 1, 2, 3, 4, 5, 9]
print(numbers)

reversed_numbers = sorted(numbers, reverse=True) # [9, 5, 4, 3, 2, 1, 1]
print(reversed_numbers)

numbers.reverse() # [9, 5, 4, 3, 2, 1, 1]
print(numbers)

new_list = numbers.copy()
print(new_list)
```

القواميس (Dictionaries) - عمليات متقدمة

القواميس هي مجموعات من أزواج المفتاح-القيمة. المفاتيح يجب أن تكون فريدة وغير قابلة للتغيير (مثل السلاسل النصية والأعداد والصفوف).

1. الوصول إلى العناصر:

- `dict[key]` : للوصول إلى قيمة المفتاح. سيؤدي إلى خطأ إذا لم يكن المفتاح موجودًا.
- `dict.get(key, default_value)` : للوصول إلى قيمة المفتاح. إذا لم يكن المفتاح موجودًا، تُرجع `None` أو `default_value` إذا تم توفيره.

```
 person = {"name": "أحمد", "age": 30, "city": "القاهرة"}
print(person["name"]) # أحمد

print(person.get("age")) # 30
print(person.get("country", "غير معروف")) # غير معروف
```

2. إضافة وتعديل عناصر:

- `dict[key] = value` : لإضافة زوج مفتاح-قيمة جديد أو تعديل قيمة مفتاح موجود.

```
 person["email"] = "ahmed@example.com" # إضافة عنصر جديد
print(person)

person["age"] = 31 # تعديل قيمة عنصر موجود
print(person)
```

3. إزالة عناصر:

- `del dict[key]` : تُزيل زوج المفتاح-القيمة المحدد.
- `dict.pop(key, default_value)` : تُزيل المفتاح وتُرجع قيمته. يمكن توفير قيمة افتراضية إذا لم يكن المفتاح موجودًا.
- `dict.popitem()` : تُزيل وتُرجع زوج مفتاح-قيمة عشوائي (في الإصدارات الحديثة من بايثون، تُزيل آخر عنصر تم إدخاله).
- `dict.clear()` : تُزيل جميع العناصر من القاموس.

```python
person = {"name": "أحمد", "age": 30, "city": "القاهرة"}
del person["city"]
print(person)

email = person.pop("email", "لا يوجد بريد إلكتروني")
print(email, person)

item = person.popitem()
print(item, person)

person.clear()
print(person)
```

4. طرق عرض القاموس:

- `keys()`: تُرجع كائن عرض (view object) لجميع المفاتيح في القاموس.
- `values()`: تُرجع كائن عرض لجميع القيم في القاموس.
- `items()`: تُرجع كائن عرض لجميع أزواج المفتاح-القيمة كصفوف (tuples).

```python
person = {"name": "أحمد", "age": 30, "city": "القاهرة"}
print(person.keys())
print(person.values())
print(person.items())

for key in person.keys():
    print(key)

for value in person.values():
    print(value)

for key, value in person.items():
    print(f"{key}: {value}")
```

المجموعات (Sets) - عمليات متقدمة

المجموعات هي مجموعات غير مرتبة من العناصر الفريدة. تُستخدم بشكل شائع لإزالة التكرارات وإجراء عمليات المجموعات الرياضية (مثل الاتحاد والتقاطع).

1. إنشاء المجموعات:

- باستخدام الأقواس المتعرجة `{}` (للمجموعات غير الفارغة).
- باستخدام الدالة `set()` (لإنشاء مجموعة فارغة أو من كائن قابل للتكرار).

```
my_set = {1, 2, 3, 2}
print(my_set) # {1, 2, 3}

empty_set = set()
print(empty_set)

list_to_set = set([1, 2, 2, 3, 4, 4])
print(list_to_set) # {1, 2, 3, 4}
```

2. إضافة وإزالة عناصر:

- `add(item)`: تُضيف عنصرًا واحدًا إلى المجموعة.
- `remove(item)`: تُزيل العنصر المحدد. سيؤدي إلى خطأ إذا لم يكن العنصر موجودًا.
- `discard(item)`: تُزيل العنصر المحدد إذا كان موجودًا، ولا تفعل شيئًا إذا لم يكن موجودًا (لا تُحدث خطأ).
- `pop()`: تُزيل وتُرجع عنصرًا عشوائيًا من المجموعة.
- `clear()`: تُزيل جميع العناصر من المجموعة.

```
my_set = {1, 2, 3}
my_set.add(4) # {1, 2, 3, 4}
print(my_set)

my_set.remove(2) # {1, 3, 4}
print(my_set)

my_set.discard(5) # لا يحدث خطأ
print(my_set)

popped_item = my_set.pop()
print(popped_item, my_set)

my_set.clear()
print(my_set)
```

3. عمليات المجموعات الرياضية:

- `union()` أو `|`: الاتحاد (جميع العناصر الفريدة من كلا المجموعتين).
- `intersection()` أو `&`: التقاطع (العناصر المشتركة بين المجموعتين).
- `difference()` أو `-`: الفرق (العناصر الموجودة في المجموعة الأولى وليست في الثانية).

- `symmetric_difference()` أو `^` : الفرق المتماثل (العناصر الموجودة في إحدى المجموعتين ولكن ليست في كليهما).
- `issubset()` : هل المجموعة فرعية من أخرى؟
- `issuperset()` : هل المجموعة تحتوي على أخرى؟

```
 set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

print(set1.union(set2)) # {1, 2, 3, 4, 5, 6}
print(set1 | set2)      # {1, 2, 3, 4, 5, 6}

print(set1.intersection(set2)) # {3, 4}
print(set1 & set2)             # {3, 4}

print(set1.difference(set2)) # {1, 2}
print(set1 - set2)           # {1, 2}

print(set1.symmetric_difference(set2)) # {1, 2, 5, 6}
print(set1 ^ set2)                     # {1, 2, 5, 6}

print({1, 2}.issubset(set1)) # True
print(set1.issuperset({1, 2})) # True
```

تمارين الفصل الرابع

التمرين 1:

لديك قائمة من الأرقام: `numbers = [10, 20, 30, 20, 40, 50]`. * أضف الرقم `60` إلى نهاية القائمة. * أدخل الرقم `15` في الفهرس `1`. * أزل أول ظهور للرقم `20`. * أزل العنصر الأخير من القائمة واطبعه. * اطبع القائمة النهائية.

التمرين 2:

لديك قاموس يمثل معلومات طالب: `student = {"name": "ليلى", "id": "12345", "major": "علوم حاسب"}`. * أضف مفتاحًا جديدًا `"gpa"` بقيمة `3.8`. * غير قيمة `"major"` إلى `"هندسة برمجيات"`. * اطبع قيمة `"id"` باستخدام `get()`. * أزل المفتاح `"gpa"` من القاموس. * اطبع القاموس النهائي.

التمرين 3:

لديك مجموعتان: `set_a = {1, 2, 3, 4, 5}` و `set_b = {4, 5, 6, 7, 8}`. * احسب الاتحاد بين المجموعتين واطبعه. * احسب التقاطع بين المجموعتين واطبعه. * احسب الفرق بين

`set_a` و `set_b` واطبعه. * احسب الفرق المتماثل بين المجموعتين واطبعه.

حلول تمارين الفصل الرابع

حل التمرين 1:

```
numbers = [10, 20, 30, 20, 40, 50]

numbers.append(60)
print("بعد إضافة 60:", numbers)

numbers.insert(1, 15)
print("بعد إدخال 15 في الفهرس 1:", numbers)

numbers.remove(20)
print("بعد إزالة أول 20:", numbers)

last_item = numbers.pop()
print("العنصر الأخير الذي تم إزالته:", last_item)
print("القائمة النهائية:", numbers)
```

حل التمرين 2:

```
student = {"name": "ليلى", "id": "12345", "major": "علوم حاسب"}

student["gpa"] = 3.8
print("بعد إضافة GPA:", student)

student["major"] = "هندسة برمجيات"
print("بعد تغيير التخصص:", student)

student_id = student.get("id")
print("رقم الطالب:", student_id)

del student["gpa"]
print("القاموس النهائي:", student)
```

حل التمرين 3:

```
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}

union_set = set_a.union(set_b)
print("الاتحاد:", union_set)

intersection_set = set_a.intersection(set_b)
print("التقاطع:", intersection_set)

difference_set = set_a.difference(set_b)
print("الفرق (set_a - set_b):", difference_set)

symmetric_difference_set = set_a.symmetric_difference(set_b)
print("الفرق المتماثل:", symmetric_difference_set)
```

الفصل الخامس: التعامل مع الملفات

يُعد التعامل مع الملفات جزءًا أساسيًا من العديد من البرامج، حيث يسمح لنا بقراءة البيانات من مصادر خارجية أو حفظ النتائج لاستخدامها لاحقًا. في بايثون، يمكننا التعامل مع أنواع مختلفة من الملفات، مثل الملفات النصية، وملفات CSV، وغيرها.

قراءة وكتابة الملفات النصية

لفتح ملف في بايثون، نستخدم الدالة المدمجة `open()`. تأخذ هذه الدالة اسم الملف ووضع الفتح (mode) كمعاملات.

أوضاع الفتح (File Modes):

- `"r"` : للقراءة (افتراضي). يرفع خطأ إذا لم يكن الملف موجودًا.
- `"w"` : للكتابة. ينشئ ملفًا جديدًا إذا لم يكن موجودًا، ويمحو محتويات الملف إذا كان موجودًا.
- `"a"` : للإلحاق (append). ينشئ ملفًا جديدًا إذا لم يكن موجودًا، ويضيف المحتوى إلى نهاية الملف إذا كان موجودًا.
- `"x"` : للإنشاء الحصري. ينشئ ملفًا جديدًا، ويرفع خطأ إذا كان الملف موجودًا بالفعل.
- `"t"` : للوضع النصي (افتراضي).
- `"b"` : للوضع الثنائي (binary).

1. فتح وإغلاق الملفات:

من المهم جدًا إغلاق الملف بعد الانتهاء من التعامل معه لتحرير الموارد. يمكن القيام بذلك باستخدام الدالة `close()`.

```
# الكتابة إلى ملف
file = open("my_file.txt", "w")
file.write("مرحباً بكم في عالم بايثون.\n")
file.write("هذا سطر جديد.\n")
file.close()

# القراءة من ملف
file = open("my_file.txt", "r")
content = file.read()
print(content)
file.close()
```

2. استخدام statement `with` (الطريقة المفضلة):

تُعد statement `with` الطريقة المفضلة للتعامل مع الملفات في بايثون. تضمن هذه الطريقة إغلاق الملف تلقائيًا، حتى لو حدث خطأ أثناء التعامل مع الملف.

```
# الكتابة إلى ملف باستخدام with
with open("my_file.txt", "w") as file:
    file.write("مرحباً بكم في عالم بايثون.\n")
    file.write("هذا سطر جديد.\n")

# القراءة من ملف باستخدام with
with open("my_file.txt", "r") as file:
    content = file.read()
    print(content)
```

3. قراءة الملفات سطرًا سطرًا:

يمكن قراءة الملفات سطرًا سطرًا باستخدام حلقة `for`، وهو أمر مفيد للملفات الكبيرة.

```
with open("my_file.txt", "r") as file:
    for line in file:
        print(line.strip()) # .strip() لإزالة المسافات البيضاء الزائدة و newline
```

4. الإلحاق إلى ملف (Appending):

لإضافة محتوى إلى نهاية ملف موجود دون مسح محتوياته الأصلية، نستخدم وضع `"a"`.

```python
with open("my_file.txt", "a") as file:
    file.write("هذا سطر مضاف جديد.\n")

with open("my_file.txt", "r") as file:
    content = file.read()
    print(content)
```

التعامل مع الأخطاء (Error Handling)

عند التعامل مع الملفات، قد تحدث أخطاء (استثناءات) مثل محاولة فتح ملف غير موجود. لمعالجة هذه الأخطاء بشكل أنيق، نستخدم كتل `try`, `except`, `finally`.

- `try`: الكتلة التي تحتوي على الكود الذي قد يسبب خطأ.
- `except`: الكتلة التي يتم تنفيذها إذا حدث خطأ من نوع معين داخل كتلة `try`.
- `finally`: الكتلة التي يتم تنفيذها دائمًا، سواء حدث خطأ أم لا. تُستخدم عادة لتنظيف الموارد (مثل إغلاق الملفات).

```python
try:
    with open("non_existent_file.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("خطأ: الملف غير موجود.")
except Exception as e:
    print(f"حدث خطأ آخر: {e}")
finally:
    print("انتهت محاولة التعامل مع الملف.")
```

مثال يوضح استخدام `else` مع `try-except`:

يمكن استخدام `else` مع `try-except` لتنفيذ كود معين إذا لم تحدث أي استثناءات في كتلة `try`.

```python
try:
    with open("my_file.txt", "r") as file:
        content = file.read()
        print("تم قراءة الملف بنجاح.")
except FileNotFoundError:
    print("خطأ: الملف غير موجود.")
else:
    print("محتوى الملف:\n", content)
finally:
    print("انتهت عملية التعامل مع الملف.")
```

تمارين الفصل الخامس

التمرين 1:

اكتب برنامجًا يقوم بإنشاء ملف نصي جديد باسم `greetings.txt` ويحتوي على السطرين التاليين: "أهلاً وسهلاً في بايثون! تعلم البرمجة ممتع."

التمرين 2:

اكتب برنامجًا يقرأ محتويات الملف `greetings.txt` الذي أنشأته في التمرين السابق ويطبع كل سطر على حدة.

التمرين 3:

اكتب برنامجًا يضيف السطر "هذا سطر جديد مضاف." إلى نهاية الملف `greetings.txt` دون مسح المحتوى الأصلي. ثم اقرأ الملف بالكامل واطبعه.

التمرين 4:

اكتب برنامجًا يحاول فتح ملف باسم `non_existent.txt` للقراءة. استخدم `try-except` لالتقاط الخطأ `FileNotFoundError` وطباعة رسالة مناسبة إذا لم يتم العثور على الملف.

حلول تمارين الفصل الخامس

حل التمرين 1:

```
with open("greetings.txt", "w") as file:
    file.write("أهلاً وسهلاً في بايثون!\n")
    file.write("تعلم البرمجة ممتع.\n")
print("تم إنشاء الملف greetings.txt بنجاح.")
```

حل التمرين 2:

```
try:
    with open("greetings.txt", "r") as file:
        for line in file:
            print(line.strip())
except FileNotFoundError:
    print("الملف greetings.txt غير موجود.")
```

حل التمرين 3:

```
with open("greetings.txt", "a") as file:
    file.write("هذا سطر جديد مضاف.\n")
print("تم إضافة سطر جديد إلى greetings.txt.")

print("محتوى الملف بعد الإضافة:")
with open("greetings.txt", "r") as file:
    content = file.read()
    print(content)
```

حل التمرين 4:

```
try:
    with open("non_existent.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("خطأ: الملف non_existent.txt غير موجود. يرجى التأكد من وجوده.")
```

الفصل السادس: البرمجة الكائنية التوجه (Object-Oriented Programming - OOP)

البرمجة الكائنية التوجه (OOP) هي نموذج برمجي يعتمد على مفهوم "الكائنات"، والتي يمكن أن تحتوي على بيانات (خصائص) ورموز (دوال). الهدف الرئيسي من OOP هو تنظيم الكود وجعله أكثر قابلية لإعادة الاستخدام، والصيانة، والتوسع. بايثون تدعم OOP بشكل كامل.

المفاهيم الأساسية

1. الفئات (Classes):

الفئة هي مخطط (blueprint) لإنشاء الكائنات. تُعرف الفئة الخصائص (attributes) التي ستمتلكها الكائنات والدوال (methods) التي يمكن للكائنات تنفيذها.

```python
class Dog:
    # خاصية الفئة (تنطبق على جميع الكائنات من هذه الفئة)
    species = "Canis familiaris"

    # دالة البناء (Constructor) - تُستدعى عند إنشاء كائن جديد
    def __init__(self, name, age):
        self.name = name  # خصائص الكائن (Instance attributes)
        self.age = age

    # دالة (Method) - سلوك الكائن
    def bark(self):
        return f"{self.name} يقول: واف واف!"

    def get_age_in_dog_years(self):
        return self.age * 7
```

2. الكائنات (Objects):

الكائن هو نسخة (instance) من الفئة. يمكنك إنشاء العديد من الكائنات من نفس الفئة، وكل كائن سيكون له خصائصه الخاصة.

```python
# إنشاء كائنات (instances) من الفئة Dog
my_dog = Dog("لوسي", 3)
your_dog = Dog("ماكس", 5)

# الوصول إلى خصائص الكائنات
print(my_dog.name)  # لوسي
print(your_dog.age) # 5

# استدعاء دوال الكائنات
print(my_dog.bark()) # لوسي يقول: واف واف!
print(your_dog.get_age_in_dog_years()) # 35

# الوصول إلى خاصية الفئة
print(Dog.species) # Canis familiaris
print(my_dog.species) # Canis familiaris
```

الوراثة (Inheritance)

الوراثة هي آلية تسمح لفئة (الفئة الفرعية أو المشتقة) باكتساب الخصائص والدوال من فئة أخرى (الفئة الأساسية أو الأصل). هذا يعزز إعادة استخدام الكود.

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("يجب على الفئات الفرعية تنفيذ هذه الدالة")

class Cat(Animal):
    def __init__(self, name, breed):
        super().__init__(name) # استدعاء دالة البناء للفئة الأصل
        self.breed = breed

    def speak(self):
        return f"{self.name} يقول: مواء!"

class Dog(Animal):
    def __init__(self, name, color):
        super().__init__(name)
        self.color = color

    def speak(self):
        return f"{self.name} يقول: واف!"

my_cat = Cat("مشمش", "سيامي")
my_dog = Dog("بندق", "بني")

print(my_cat.name)  # مشمش
print(my_cat.speak()) # مشمش يقول: مواء!

print(my_dog.name)  # بندق
print(my_dog.speak()) # بندق يقول: واف!
```

التغليف (Encapsulation)

التغليف هو مبدأ تجميع البيانات (الخصائص) والدوال (الأساليب) التي تعمل على تلك البيانات داخل وحدة واحدة (الفئة)، وإخفاء التفاصيل الداخلية عن العالم الخارجي. في بايثون، لا يوجد مفهوم صارم للوصول الخاص (private) أو العام (public) كما في بعض اللغات الأخرى، ولكن هناك اصطلاحات:

- **خصائص عامة (Public attributes):** يمكن الوصول إليها وتعديلها مباشرة من خارج الفئة (مثال: `obj.attribute`).
- **خصائص محمية (Protected attributes):** تبدأ بشرطة سفلية واحدة `_` (مثال: `_attribute`). هذا اصطلاح يشير إلى أنه يجب التعامل معها بحذر، ولكن لا يمنع الوصول إليها تقنيًا.

- **خصائص خاصة (Private attributes):** تبدأ بشرطتين سفليتين `__` (مثال: `__attribute`). بايثون تقوم بتغيير اسمها (name mangling) لجعل الوصول إليها من الخارج أكثر صعوبة، ولكن ليس مستحيلاً.

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance # خاصية خاصة (اصطلاحاً)

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"تم الإيداع. الرصيد الجديد: {self.__balance}")
        else:
            print("مبلغ الإيداع يجب أن يكون موجباً.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"تم السحب. الرصيد الجديد: {self.__balance}")
        else:
            print("مبلغ السحب غير صالح أو الرصيد غير كافٍ.")

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
# print(account.__balance) # سيؤدي إلى خطأ (AttributeError) بسبب name mangling
print(account.get_balance()) # الطريقة الصحيحة للوصول إلى الرصيد
```

التجريد (Abstraction)

التجريد هو عملية إخفاء تفاصيل التنفيذ المعقدة وعرض الوظائف الأساسية فقط للمستخدم. في بايثون، يمكن تحقيق التجريد باستخدام الفئات المجردة (Abstract Classes) والدوال المجردة (Abstract Methods) من خلال وحدة (Abstract Base Classes) `abc`.

الفئة المجردة هي فئة لا يمكن إنشاء كائنات منها مباشرة، ويجب على الفئات الفرعية التي ترث منها تنفيذ دوالها المجردة.

```python
from abc import ABC, abstractmethod

class Shape(ABC): # فئة مجردة
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14 * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# shape = Shape() # سيؤدي إلى خطأ: Can't instantiate abstract class Shape

circle = Circle(5)
print(f"مساحة الدائرة: {circle.area()}")
print(f"محيط الدائرة: {circle.perimeter()}")

rectangle = Rectangle(4, 6)
print(f"مساحة المستطيل: {rectangle.area()}")
print(f"محيط المستطيل: {rectangle.perimeter()}")
```

تمارين الفصل السادس

التمرين 1:

أنشئ فئة باسم `Car` (سيارة) تحتوي على الخصائص التالية: `make` (الشركة المصنعة)، `model` (الموديل)، `year` (السنة). أضف دالة باسم `display_info` تطبع معلومات السيارة (الشركة المصنعة، الموديل، السنة). أنشئ كائنين من الفئة `Car` واعرض معلوماتهما.

التمرين 2:

أنشئ فئة أساسية باسم `Person` (شخص) تحتوي على خاصية `name` ودالة `introduce` تطبع "مرحباً، أنا [الاسم]". أنشئ فئة فرعية باسم `Student` (طالب) ترث من `Person`. أضف خاصية جديدة `student_id` (رقم الطالب) إلى `Student`. اجعل دالة `introduce` في `Student` تطبع "مرحباً، أنا [الاسم] ورقم طالبي هو [رقم الطالب]". أنشئ كائنًا من `Student` واعرض معلوماته.

التمرين 3:

أنشئ فئة باسم `Product` (منتج) تحتوي على خاصيتين خاصتين (باستخدام اصطلاح `__`): `__name` و `__price`. أضف دوال عامة (public methods) للوصول إلى هذه الخصائص (`get_name`, `get_price`) وتعديلها (`set_price`). أنشئ كائنًا من `Product`، ثم حاول الوصول إلى الخصائص الخاصة مباشرة، ولاحظ الخطأ. ثم استخدم الدوال العامة للوصول والتعديل.

حلول تمارين الفصل السادس

حل التمرين 1:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        print(f"الشركة المصنعة: {self.make}, الموديل: {self.model}, السنة: {self.year}")

car1 = Car("تويوتا", "كامري", 2020)
car2 = Car("هوندا", "سيفيك", 2022)

car1.display_info()
car2.display_info()
```

حل التمرين 2:

```python
 class Person:
    def __init__(self, name):
        self.name = name

    def introduce(self):
        print(f"مرحباً، أنا {self.name}.")

class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)
        self.student_id = student_id

    def introduce(self):
        print(f"مرحباً، أنا {self.name} ورقم طالبي هو {self.student_id}.")

student1 = Student("فاطمة", "S12345")
student1.introduce()
```

حل التمرين 3:

```python
class Product:
    def __init__(self, name, price):
        self.__name = name
        self.__price = price

    def get_name(self):
        return self.__name

    def get_price(self):
        return self.__price

    def set_price(self, new_price):
        if new_price > 0:
            self.__price = new_price
            print(f"تم تحديث سعر {self.__name} إلى {self.__price}")
        else:
            print("السعر الجديد يجب أن يكون موجباً.")

product1 = Product("هاتف ذكي", 1500)

# محاولة الوصول المباشر (سيؤدي إلى خطأ)
# print(product1.__name)

# استخدام الدوال العامة للوصول
print(f"اسم المنتج: {product1.get_name()}")
print(f"سعر المنتج: {product1.get_price()}")

# استخدام الدوال العامة للتعديل
product1.set_price(1450)
print(f"السعر بعد التعديل: {product1.get_price()}")
product1.set_price(-100) # مثال على قيمة غير صالحة
```

الفصل السابع: الوحدات والحزم (Modules and Packages)

مع نمو برامجك، قد يصبح الكود كبيرًا ومعقدًا. هنا يأتي دور الوحدات (Modules) والحزم (Packages) في بايثون. تسمح لك هذه المفاهيم بتنظيم الكود الخاص بك في ملفات ومجلدات منفصلة، مما يجعله أكثر قابلية للإدارة، وإعادة الاستخدام، والفهم.

1. الوحدات (Modules)

الوحدة هي ببساطة ملف بايثون (`.py`) يحتوي على تعريفات ودوال وفئات ومتغيرات. يمكنك استخدام أي ملف بايثون كوحدة.

استيراد الوحدات:

لاستخدام الكود الموجود في وحدة أخرى، يجب عليك استيرادها باستخدام الكلمة المفتاحية `import`.

مثال: لنفترض أن لدينا ملفًا باسم `my_module.py` يحتوي على الدالة التالية:

```
# my_module.py
def greet(name):
    return f"أهلاً يا {name} من وحدتي!"

PI = 3.14159
```

يمكنك استيراد واستخدام هذه الوحدة في ملف بايثون آخر:

```
# main.py
import my_module

print(my_module.greet("علي"))
print(my_module.PI)
```

طرق أخرى للاستيراد:

- `from module import name1, name2`: لاستيراد أسماء محددة (دوال، متغيرات، فئات) مباشرة من الوحدة. هذا يسمح لك باستخدامها دون الحاجة إلى بادئة اسم الوحدة.

  ```python

  # main.py

  from my_module import greet, PI

  print(greet("فاطمة")) print(PI) ```

- `from module import *`: لاستيراد جميع الأسماء من الوحدة. لا يُنصح بهذا في البرامج الكبيرة لأنه قد يؤدي إلى تضارب في الأسماء.

  ```python

# main.py

* from my_module import

``` print(PI) (("خالد")greet)print

- **`import module as alias`**: لتعيين اسم مستعار (alias) للوحدة، مما يجعل الكود أقصر وأسهل في القراءة.

python```

# main.py

import my_module as mm

``` print(mm.PI) (("ليلى")mm.greet)print

### 2. إنشاء الوحدات الخاصة بك

كما رأينا في المثال أعلاه، كل ما عليك فعله هو حفظ الكود الخاص بك في ملف `.py`، ثم يمكنك استيراده في ملفات أخرى.

### 3. الحزم (Packages)

الحزمة هي طريقة لتنظيم الوحدات ذات الصلة في بنية دليل هرمية. الحزمة هي في الأساس مجلد يحتوي على ملف `__init__.py` (يمكن أن يكون فارغًا) وملفات وحدات أخرى أو حزم فرعية.

**هيكل الحزمة:**

```
my_package/
├── __init__.py
├── module_a.py
├── module_b.py
└── sub_package/
    ├── __init__.py
    └── module_c.py
```

مثال على الاستيراد من حزمة:

لنفرض أن `module_a.py` يحتوي على دالة `func_a` و `module_c.py` يحتوي على دالة `func_c`.

```
# main.py
from my_package import module_a
from my_package.sub_package import module_c

print(module_a.func_a())
print(module_c.func_c())
```

4. إدارة الحزم (pip)

`pip` هو مدير الحزم القياسي لبايثون. يُستخدم لتثبيت وإدارة حزم بايثون الخارجية (المكتبات) التي لا تأتي مدمجة مع بايثون.

أوامر `pip` الشائعة:

* **تثبيت حزمة:** `bash pip install package_name` مثال: `pip install requests` (لتثبيت مكتبة Requests للتعامل مع طلبات HTTP).
* **تثبيت إصدار محدد:** `bash pip install package_name==version_number` مثال: `pip install requests==2.28.1`
* **ترقية حزمة:** `bash pip install --upgrade package_name`
* **إلغاء تثبيت حزمة:** `bash pip uninstall package_name`
* **عرض الحزم المثبتة:** `bash pip list`
* **حفظ بيئة المشروع (requirements.txt):** من الممارسات الجيدة حفظ قائمة بالحزم التي يعتمد عليها مشروعك في ملف `requirements.txt`. هذا يسمح للآخرين (أو لنفسك في المستقبل) بتثبيت جميع التبعيات بسهولة.

  `bash pip freeze > requirements.txt`
* **تثبيت الحزم من `requirements.txt`:** `bash pip install -r requirements.txt`

تمارين الفصل السابع

التمرين 1:

أنشئ ملفًا باسم `math_operations.py` يحتوي على دالتين: * `add(a, b)`: تُرجع مجموع `a` و `b`. * `subtract(a, b)`: تُرجع الفرق بين `a` و `b`.

ثم، في ملف بايثون منفصل، استورد الدالتين واستخدمهما لإجراء عملية جمع وطرح، واطبع النتائج.

التمرين 2:

أنشئ بنية حزمة بسيطة:

```
my_project/
├── main.py
└── utils/
    ├── __init__.py
    └── string_utils.py
```

في `string_utils.py`، اكتب دالة باسم `reverse_string(s)` تُرجع السلسلة النصية معكوسة. في `main.py`، استورد الدالة `reverse_string` من الحزمة `utils` واستخدمها لعكس سلسلة نصية من اختيارك، ثم اطبع النتيجة.

التمرين 3:

افترض أنك تريد استخدام مكتبة `colorama` لتلوين النصوص في الطرفية. استخدم `pip` لتثبيت هذه المكتبة. (ملاحظة: لا تحتاج إلى كتابة كود بايثون لهذا التمرين، فقط استخدم أمر `pip`.)

حلول تمارين الفصل السابع

حل التمرين 1:

`math_operations.py`:

```python
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

ملف بايثون منفصل (مثال: `main_app.py`):

```python
from math_operations import add, subtract

result_add = add(10, 5)
print(f"الجمع: {result_add}")

result_subtract = subtract(10, 5)
print(f"الطرح: {result_subtract}")
```

حل التمرين 2:

`my_project/utils/string_utils.py`:

```
def reverse_string(s):
    return s[::-1]
```

`my_project/main.py`:

```
from utils.string_utils import reverse_string

text = "بايثون"
reversed_text = reverse_string(text)
print(f"النص الأصلي: {text}")
print(f"النص المعكوس: {reversed_text}")
```

حل التمرين 3:

في الطرفية (Terminal):

```
pip install colorama
```

الفصل الثامن: التعامل مع الأخطاء والاستثناءات

في عالم البرمجة، الأخطاء والاستثناءات جزء لا يتجزأ من عملية التطوير. لا يوجد برنامج مثالي لا يحتوي على أخطاء. تعلم كيفية التعامل مع هذه الأخطاء بفعالية أمر بالغ الأهمية لإنشاء برامج قوية وموثوقة. في بايثون، تُعرف الأخطاء التي تحدث أثناء تنفيذ البرنامج بالاستثناءات (Exceptions).

1. أنواع الأخطاء الشائعة

هناك أنواع مختلفة من الأخطاء التي قد تواجهها في بايثون:

- **أخطاء بناء الجملة (Syntax Errors):** تحدث عندما لا يتبع الكود قواعد بناء الجملة للغة بايثون. يكتشفها المفسر قبل تشغيل البرنامج.

  ```python
  ```

مثال على خطأ بناء الجملة

print("مرحباً" # قوس مفقود

```

* **الاستثناءات (Exceptions):** تحدث أثناء تنفيذ البرنامج عندما يواجه المفسر موقفًا لا يمكنه التعامل معه. على عكس أخطاء بناء الجملة، فإن الاستثناءات لا تمنع البرنامج من البدء، ولكنها توقف تنفيذه إذا لم يتم التعامل معها.

بعض الاستثناءات الشائعة: * `NameError`: عند محاولة استخدام متغير أو دالة غير معرفة. * `TypeError`: عند محاولة إجراء عملية على نوع بيانات غير مناسب. * `ValueError`: عند محاولة تحويل قيمة إلى نوع غير مناسب (مثل تحويل نص غير رقمي إلى عدد). * `ZeroDivisionError`: عند محاولة القسمة على صفر. * `IndexError`: عند محاولة الوصول إلى فهرس غير موجود في قائمة أو سلسلة نصية. * `KeyError`: عند محاولة الوصول إلى مفتاح غير موجود في قاموس. * `FileNotFoundError`: عند محاولة فتح ملف غير موجود.

```python

## أمثلة على الاستثناءات

---

print(x) # NameError: name 'x' is not defined

---

print("5" + 10) # TypeError: can only concatenate str (not "int") to str

---

int("hello") # ValueError: invalid 'literal for int() with base 10: 'hello

---

print(10 / 0) # ZeroDivisionError: division by zero

---

my_list = [1, 2]

---

print(my_list[2]) # IndexError: list

## index out of range

## = my_dict

## 'print(my_dict["b"]) # KeyError: 'b

```

2. معالجة الاستثناءات (Exception Handling)

لمنع البرنامج من الانهيار عند حدوث استثناء، يمكننا استخدام كتل `finally` ,`else` ,`except` ,`try`.

`try` و `except`:

- `try`: ضع الكود الذي قد يسبب استثناءً داخل هذه الكتلة.
- `except`: إذا حدث استثناء داخل كتلة `try`، يتم تنفيذ الكود الموجود في كتلة `except` المطابقة لنوع الاستثناء.

```
try:
    num1 = int(input("أدخل الرقم الأول: "))
    num2 = int(input("أدخل الرقم الثاني: "))
    result = num1 / num2
    print(f"النتيجة: {result}")
except ValueError:
    print("خطأ: يرجى إدخال أرقام صحيحة فقط.")
except ZeroDivisionError:
    print("خطأ: لا يمكن القسمة على صفر.")
except Exception as e: # التقاط أي استثناء آخر
    print(f"حدث خطأ غير متوقع: {e}")
```

يمكنك تحديد عدة كتل `except` لأنواع مختلفة من الاستثناءات. إذا لم يتم تحديد نوع الاستثناء بعد `except`، فإنه سيلتقط أي نوع من الاستثناءات.
```

:else

تُنفذ كتلة `else` إذا لم تحدث أي استثناءات في كتلة `try`.

```
try:
    num = int(input("أدخل رقماً: "))
except ValueError:
    print("هذا ليس رقماً صحيحاً.")
else:
    print(f"لقد أدخلت الرقم: {num}")
    print("لا توجد أخطاء.")
```

:finally

تُنفذ كتلة `finally` دائمًا، بغض النظر عما إذا حدث استثناء أم لا. تُستخدم عادة لتنظيف الموارد (مثل إغلاق الملفات أو اتصالات قاعدة البيانات).

```
try:
    file = open("my_data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("الملف غير موجود.")
finally:
    if 'file' in locals() and not file.closed: # التحقق مما إذا كان الملف مفتوحاً
        file.close()
        print("تم إغلاق الملف.")
```

رفع الاستثناءات (Raising Exceptions):

يمكنك أيضًا رفع استثناءات خاصة بك باستخدام الكلمة المفتاحية `raise`. هذا مفيد عندما تريد فرض شرط معين في الكود الخاص بك.

```python
def validate_age(age):
    if not isinstance(age, int):
        raise TypeError("العمر يجب أن يكون عدداً صحيحاً.")
    if age < 0 or age > 120:
        raise ValueError("العمر يجب أن يكون بين 0 و 120.")
    print("العمر صالح.")

try:
    validate_age(30)
    validate_age(-5)
except (TypeError, ValueError) as e:
    print(f"خطأ في التحقق من العمر: {e}")
```

تمارين الفصل الثامن

التمرين 1:

اكتب برنامجًا يطلب من المستخدم إدخال رقمين. قم بإجراء عملية القسمة بينهما. استخدم `try-except` لالتقاط الأخطاء المحتملة مثل `ValueError` (إذا أدخل المستخدم نصًا بدلاً من رقم) و `ZeroDivisionError` (إذا حاول القسمة على صفر). اطبع رسالة مناسبة لكل نوع من الأخطاء.

التمرين 2:

اكتب دالة باسم `get_list_element` تأخذ قائمة وفهرسًا كمعاملات. حاول الوصول إلى العنصر في الفهرس المحدد. استخدم `try-except` لالتقاط `IndexError` إذا كان الفهرس خارج النطاق، واطبع رسالة مناسبة. إذا لم يحدث خطأ، اطبع العنصر.

التمرين 3:

اكتب دالة باسم `process_data` تأخذ قائمة من الأرقام. داخل الدالة، استخدم `try-except-else-finally`. * في `try`: حاول حساب متوسط الأرقام في القائمة. * في `except TypeError`: إذا كانت القائمة تحتوي على عناصر غير رقمية، اطبع "خطأ: القائمة تحتوي على عناصر غير رقمية." * في `except ZeroDivisionError`: إذا كانت القائمة فارغة، اطبع "خطأ: لا يمكن حساب المتوسط لقائمة فارغة." * في `else`: إذا تم حساب المتوسط بنجاح، اطبع "تم حساب المتوسط بنجاح: [المتوسط]." * في `finally`: اطبع "انتهت عملية معالجة البيانات."

اختبر الدالة مع قائمة أرقام، قائمة فارغة، وقائمة تحتوي على نص.

حلول تمارين الفصل الثامن

حل التمرين 1:

```python
try:
    num1 = float(input("أدخل الرقم الأول: "))
    num2 = float(input("أدخل الرقم الثاني: "))
    result = num1 / num2
    print(f"النتيجة: {result}")
except ValueError:
    print("خطأ: يرجى إدخال أرقام صالحة فقط.")
except ZeroDivisionError:
    print("خطأ: لا يمكن القسمة على صفر.")
except Exception as e:
    print(f"حدث خطأ غير متوقع: {e}")
```

حل التمرين 2:

```python
def get_list_element(my_list, index):
    try:
        element = my_list[index]
        print(f"العنصر في الفهرس {index} هو: {element}")
    except IndexError:
        print(f"خطأ: الفهرس {index} خارج نطاق القائمة.")

my_numbers = [10, 20, 30, 40, 50]
get_list_element(my_numbers, 2)  # عنصر موجود
get_list_element(my_numbers, 5)  # فهرس خارج النطاق
get_list_element(my_numbers, -1) # عنصر موجود (من النهاية)
```

حل التمرين 3:

```python
def process_data(numbers):
    try:
        total = sum(numbers)
        average = total / len(numbers)
    except TypeError:
        print("خطأ: القائمة تحتوي على عناصر غير رقمية.")
    except ZeroDivisionError:
        print("خطأ: لا يمكن حساب المتوسط لقائمة فارغة.")
    else:
        print(f"تم حساب المتوسط بنجاح: {average}")
    finally:
        print("انتهت عملية معالجة البيانات.")

process_data([1, 2, 3, 4, 5]) # قائمة أرقام
process_data([])              # قائمة فارغة
process_data([1, 2, 'a', 4]) # قائمة تحتوي على نص
```

الفصل التاسع: مقدمة إلى المكتبات الشائعة

تكمن قوة بايثون الحقيقية في نظامها البيئي الغني بالمكتبات (Libraries) والحزم (Packages) التي توفر وظائف جاهزة لمجموعة واسعة من المهام. في هذا الفصل، سنقدم مقدمة سريعة لبعض المكتبات الأكثر شيوعًا واستخدامًا في مجالات مختلفة، مثل الحوسبة العلمية وتحليل البيانات وتصورها.

1. NumPy (Numerical Python)

NumPy هي مكتبة أساسية للحوسبة العلمية في بايثون. توفر كائن مصفوفة قويًا يسمى `ndarray` (N-dimensional array) ودوال للعمل مع هذه المصفوفات بكفاءة عالية. تُستخدم NumPy على نطاق واسع في علم البيانات، والتعلم الآلي، والهندسة.

الميزات الرئيسية:

- **كائن المصفوفة `ndarray`:** مصفوفات متعددة الأبعاد أسرع وأكثر كفاءة في الذاكرة من قوائم بايثون العادية للعمليات الرقمية.
- **عمليات المصفوفات:** توفر دوال قوية لإجراء عمليات رياضية ومنطقية على المصفوفات.

مثال:

```python
import numpy as np

# إنشاء مصفوفة NumPy
arr = np.array([1, 2, 3, 4, 5])
print("المصفوفة:", arr)
print("نوع المصفوفة:", type(arr))

# عمليات رياضية على المصفوفات
arr_plus_5 = arr + 5
print("المصفوفة + 5:", arr_plus_5)

# إنشاء مصفوفة ثنائية الأبعاد
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print("المصفوفة ثنائية الأبعاد:\n", matrix)

# ضرب المصفوفات (عنصر بعنصر)
matrix_mult = matrix * 2
print("المصفوفة * 2:\n", matrix_mult)

# إنشاء مصفوفة من الأصفار أو الواحدات
zeros = np.zeros((2, 3)) # مصفوفة 2x3 من الأصفار
ones = np.ones((3, 2))   # مصفوفة 3x2 من الواحدات
print("مصفوفة الأصفار:\n", zeros)
print("مصفوفة الواحدات:\n", ones)
```

2. Pandas

Pandas هي مكتبة قوية ومرنة لتحليل البيانات ومعالجتها. توفر هياكل بيانات سهلة الاستخدام وعالية الأداء، مثل `Series` (لسلسلة بيانات أحادية البعد) و `DataFrame` (لجدول بيانات ثنائي الأبعاد)، مما يجعلها مثالية للعمل مع البيانات الجدولية.

الميزات الرئيسية:

- **`DataFrame`:** هيكل بيانات جدولي يشبه جداول البيانات في Excel أو قواعد البيانات العلائقية، مع صفوف وأعمدة.
- **معالجة البيانات:** أدوات قوية لتنظيف البيانات، وتحويلها، وتحليلها، ودمجها.

مثال:

```
import pandas as pd

# إنشاء DataFrame من قاموس
data = {
    'Name': ['علي', 'سارة', 'أحمد', 'ليلى'],
    'Age': [25, 30, 35, 28],
    'City': ['الرياض', 'جدة', 'الدمام', 'القاهرة']
}
df = pd.DataFrame(data)
print("البيانات الأصلية:\n", df)

# الوصول إلى عمود
print("\nعمود الاسم:\n", df['Name'])

# إضافة عمود جديد
df['Salary'] = [5000, 6000, 7500, 5500]
print("\nبعد إضافة عمود الراتب:\n", df)

# تصفية البيانات
filtered_df = df[df['Age'] > 28]
print("\nالبيانات المصفاة (العمر > 28):\n", filtered_df)

# حساب المتوسط لعمود
print("\nمتوسط العمر:", df['Age'].mean())
```

3. Matplotlib

Matplotlib هي مكتبة شاملة لإنشاء تصورات ثابتة ومتحركة وتفاعلية في بايثون. تُستخدم لإنشاء مجموعة واسعة من الرسوم البيانية، مثل الرسوم البيانية الخطية، والمبعثرة، والأعمدة، والمخططات الدائرية، والمدرجات التكرارية.

الميزات الرئيسية:

- **تنوع الرسوم البيانية:** تدعم أنواعًا متعددة من الرسوم البيانية.
- **التخصيص:** توفر تحكمًا دقيقًا في كل جانب من جوانب الرسم البياني.

مثال (يتطلب بيئة يمكنها عرض الرسوم البيانية، مثل Jupyter Notebook أو تشغيل ملف py.):

```python
import matplotlib.pyplot as plt
import numpy as np

# رسم بياني خطي بسيط
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 6])

plt.plot(x, y)
plt.xlabel("المحور السيني")
plt.ylabel("المحور الصادي")
plt.title("رسم بياني خطي بسيط")
plt.show()

# رسم بياني مبعثر
days = [1, 2, 3, 4, 5, 6, 7]
temperature = [22, 24, 23, 25, 27, 26, 28]

plt.scatter(days, temperature)
plt.xlabel("اليوم")
plt.ylabel("درجة الحرارة")
plt.title("درجات الحرارة خلال الأسبوع")
plt.show()

# رسم بياني بالأعمدة
categories = ['A', 'B', 'C', 'D']
values = [10, 25, 15, 30]

plt.bar(categories, values)
plt.xlabel("الفئة")
plt.ylabel("القيمة")
plt.title("رسم بياني بالأعمدة")
plt.show()
```

ملاحظة: لكي تعمل أمثلة Matplotlib، قد تحتاج إلى تثبيت المكتبة باستخدام `pip install matplotlib` وتشغيل الكود في بيئة تدعم عرض الرسوم البيانية (مثل Jupyter Notebook أو IDE مع تكوين صحيح).

تمارين الفصل التاسع

التمرين 1 (NumPy):

أنشئ مصفوفة NumPy ثنائية الأبعاد (3x3) تحتوي على الأرقام من 1 إلى 9. قم بضرب جميع عناصر المصفوفة في 10. احسب مجموع كل عمود في المصفوفة واطبعه.

التمرين 2 (Pandas):

أنشئ `DataFrame` يحتوي على بيانات الطلاب التالية: | الاسم | العمر | المعدل | | :--- | :---: | :----: | | خالد | 20 | 3.5 | | نورة | 22 | 3.9 | | فهد | 21 | 3.2 | | مريم | 20 | 3.7 |

* اطبع الطلاب الذين معدلهم (المعدل) أكبر من 3.6.
* احسب متوسط عمر الطلاب واطبعه.

التمرين 3 (Matplotlib):

باستخدام Matplotlib، ارسم رسمًا بيانيًا خطيًا يمثل نمو عدد المستخدمين على مدار 5 أشهر. * الأشهر: `[1, 2, 3, 4, 5]` * عدد المستخدمين: `[100, 150, 220, 300, 400]` أضف عنوانًا للرسم البياني ومسميات للمحاور.

حلول تمارين الفصل التاسع

حل التمرين 1 (NumPy):

```
import numpy as np

matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("المصفوفة الأصلية:\n", matrix)

multiplied_matrix = matrix * 10
print("المصفوفة بعد الضرب في 10:\n", multiplied_matrix)

column_sums = np.sum(matrix, axis=0) # axis=0 يعني مجموع الأعمدة
print("مجموع كل عمود:\n", column_sums)
```

حل التمرين 2 (Pandas):

```python
import pandas as pd

data = {
    'الاسم': ['خالد', 'نورة', 'فهد', 'مريم'],
    'العمر': [20, 22, 21, 20],
    'المعدل': [3.5, 3.9, 3.2, 3.7]
}
df = pd.DataFrame(data)

# الطلاب الذين معدلهم أكبر من 3.6
high_gpa_students = df[df['المعدل'] > 3.6]
print("الطلاب ذوو المعدل العالي:\n", high_gpa_students)

# متوسط عمر الطلاب
average_age = df['العمر'].mean()
print("متوسط عمر الطلاب:", average_age)
```

حل التمرين 3 (Matplotlib):

```python
import matplotlib.pyplot as plt

months = [1, 2, 3, 4, 5]
users = [100, 150, 220, 300, 400]

plt.plot(months, users, marker='o')
plt.xlabel("الشهر")
plt.ylabel("عدد المستخدمين")
plt.title("نمو عدد المستخدمين شهرياً")
plt.grid(True)
plt.show()
```

الفصل العاشر: مشاريع تطبيقية بسيطة

بعد أن تعلمت أساسيات بايثون والمفاهيم المتقدمة، حان الوقت لتطبيق هذه المعرفة في بناء مشاريع عملية بسيطة. هذه المشاريع ستساعدك على ترسيخ فهمك، وتطوير مهاراتك في حل المشكلات، وبناء ثقتك كمبرمج.

1. مشروع آلة حاسبة بسيطة

سنقوم بإنشاء آلة حاسبة بسيطة يمكنها إجراء العمليات الأساسية: الجمع، الطرح، الضرب، والقسمة.

```python
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
        return "خطأ: لا يمكن القسمة على صفر!"
    return x / y

print("اختر العملية:")
print("1. جمع")
print("2. طرح")
print("3. ضرب")
print("4. قسمة")

while True:
    choice = input("أدخل اختيارك (1/2/3/4): ")

    if choice in ("1", "2", "3", "4"):
        try:
            num1 = float(input("أدخل الرقم الأول: "))
            num2 = float(input("أدخل الرقم الثاني: "))
        except ValueError:
            print("مدخل غير صالح. يرجى إدخال أرقام.")
            continue

        if choice == "1":
            print(f"{num1} + {num2} = {add(num1, num2)}")
        elif choice == "2":
            print(f"{num1} - {num2} = {subtract(num1, num2)}")
        elif choice == "3":
            print(f"{num1} * {num2} = {multiply(num1, num2)}")
        elif choice == "4":
            print(f"{num1} / {num2} = {divide(num1, num2)}")

        next_calculation = input("هل تريد إجراء عملية أخرى؟ (نعم/لا): ")
        if next_calculation.lower() == "لا":
            break
    else:
        print("اختيار غير صالح. يرجى إدخال 1 أو 2 أو 3 أو 4.")
```

2. مشروع لعبة تخمين الأرقام

سنقوم بإنشاء لعبة بسيطة حيث يقوم الكمبيوتر بتوليد رقم عشوائي، وعلى المستخدم تخمين هذا الرقم.

```
import random

def guess_the_number():
    secret_number = random.randint(1, 100) # يولد رقماً عشوائياً بين 1 و 100
    attempts = 0
    print("أهلاً بك في لعبة تخمين الأرقام!")
    print("لقد اخترت رقماً بين 1 و 100. حاول تخمينه.")

    while True:
        try:
            guess = int(input("أدخل تخمينك: "))
            attempts += 1

            if guess < secret_number:
                print("الرقم السري أكبر!")
            elif guess > secret_number:
                print("الرقم السري أصغر!")
            else:
                print(f"تهانينا! لقد خمنت الرقم السري ({secret_number}) في {attempts} محاولات.")
                break
        except ValueError:
            print("مدخل غير صالح. يرجى إدخال رقم صحيح.")

guess_the_number()
```

3. مشروع إدارة قائمة المهام (To-Do List)

سنقوم بإنشاء برنامج بسيط لإدارة قائمة المهام، حيث يمكن للمستخدم إضافة مهام، عرض المهام، ووضع علامة على المهام المكتملة.

```python
tasks = []

def add_task(task_name):
    tasks.append({"name": task_name, "completed": False})
    print(f"تم إضافة المهمة: {task_name}")

def view_tasks():
    if not tasks:
        print("لا توجد مهام حالياً.")
        return
    print("\nقائمة المهام:")
    for i, task in enumerate(tasks):
        status = "[مكتمل]" if task["completed"] else "[غير مكتمل]"
        print(f"{i + 1}. {task["name"]} {status}")

def complete_task(task_index):
    if 0 <= task_index < len(tasks):
        tasks[task_index]["completed"] = True
        print(f"تم وضع علامة على المهمة '{tasks[task_index]["name"]}' 
كمكتملة.")
    else:
        print("رقم مهمة غير صالح.")

while True:
    print("\nاختر خياراً:")
    print("1. إضافة مهمة")
    print("2. عرض المهام")
    print("3. إكمال مهمة")
    print("4. خروج")

    choice = input("أدخل اختيارك: ")

    if choice == "1":
        task_name = input("أدخل اسم المهمة: ")
        add_task(task_name)
    elif choice == "2":
        view_tasks()
    elif choice == "3":
        view_tasks()
        try:
            task_num = int(input("أدخل رقم المهمة لإكمالها: "))
            complete_task(task_num - 1) # -1 لأن القوائم تبدأ من 0
        except ValueError:
            print("مدخل غير صالح. يرجى إدخال رقم.")
    elif choice == "4":
        print("شكراً لاستخدامك برنامج إدارة المهام. إلى اللقاء!")
        break
    else:
        print("اختيار غير صالح. يرجى المحاولة مرة أخرى.")
```

تمارين الفصل العاشر

التمرين 1 (آلة حاسبة):

عدّل برنامج الآلة الحاسبة بحيث يدعم عملية الأس (`**`).

التمرين 2 (تخمين الأرقام):

عدّل لعبة تخمين الأرقام بحيث تمنح المستخدم عددًا محدودًا من المحاولات (مثلاً 7 محاولات). إذا لم يتمكن من التخمين في هذا العدد من المحاولات، اطبع الرقم السري.

التمرين 3 (قائمة المهام):

أضف خيارًا جديدًا إلى برنامج إدارة قائمة المهام يسمح بحذف مهمة من القائمة.

حلول تمارين الفصل العاشر

حل التمرين 1 (آلة حاسبة):

```python
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
        return "خطأ: لا يمكن القسمة على صفر!"
    return x / y

def power(x, y):
    return x ** y

print("اختر العملية:")
print("1. جمع")
print("2. طرح")
print("3. ضرب")
print("4. قسمة")
print("5. أس")

while True:
    choice = input("أدخل اختيارك (1/2/3/4/5): ")

    if choice in ("1", "2", "3", "4", "5"):
        try:
            num1 = float(input("أدخل الرقم الأول: "))
            num2 = float(input("أدخل الرقم الثاني: "))
        except ValueError:
            print("مدخل غير صالح. يرجى إدخال أرقام.")
            continue

        if choice == "1":
            print(f"{num1} + {num2} = {add(num1, num2)}")
        elif choice == "2":
            print(f"{num1} - {num2} = {subtract(num1, num2)}")
        elif choice == "3":
            print(f"{num1} * {num2} = {multiply(num1, num2)}")
        elif choice == "4":
            print(f"{num1} / {num2} = {divide(num1, num2)}")
        elif choice == "5":
            print(f"{num1} ** {num2} = {power(num1, num2)}")

        next_calculation = input("هل تريد إجراء عملية أخرى؟ (نعم/لا): ")
        if next_calculation.lower() == "لا":
            break
```

```
    else:
        print("اختيار غير صالح. يرجى إدخال 1 أو 2 أو 3 أو 4 أو 5.")
```

حل التمرين 2 (تخمين الأرقام):

```
import random

def guess_the_number_limited_attempts():
    secret_number = random.randint(1, 100)
    attempts = 0
    max_attempts = 7
    print("أهلاً بك في لعبة تخمين الأرقام!")
    print(f"لقد اخترت رقماً بين 1 و 100. لديك {max_attempts} محاولات لتخمينه.")

    while attempts < max_attempts:
        try:
            guess = int(input("أدخل تخمينك: "))
            attempts += 1

            if guess < secret_number:
                print("الرقم السري أكبر!")
            elif guess > secret_number:
                print("الرقم السري أصغر!")
            else:
                print(f"تهانينا! لقد خمنت الرقم السري ({secret_number}) في
{attempts} محاولات.")
                return
        except ValueError:
            print("مدخل غير صالح. يرجى إدخال رقم صحيح.")

    print(f"لقد نفدت محاولاتك! الرقم السري كان {secret_number}.")

guess_the_number_limited_attempts()
```

حل التمرين 3 (قائمة المهام):

```python
tasks = []

def add_task(task_name):
    tasks.append({"name": task_name, "completed": False})
    print(f"تم إضافة المهمة: {task_name}")

def view_tasks():
    if not tasks:
        print("لا توجد مهام حالياً.")
        return
    print("\nقائمة المهام:")
    for i, task in enumerate(tasks):
        status = "[مكتمل]" if task["completed"] else "[غير مكتمل]"
        print(f"{i + 1}. {task["name"]} {status}")

def complete_task(task_index):
    if 0 <= task_index < len(tasks):
        tasks[task_index]["completed"] = True
        print(f"تم وضع علامة على المهمة '{tasks[task_index]["name"]}' كمكتملة.")
    else:
        print("رقم مهمة غير صالح.")

def delete_task(task_index):
    if 0 <= task_index < len(tasks):
        removed_task = tasks.pop(task_index)
        print(f"تم حذف المهمة: '{removed_task["name"]}'.")
    else:
        print("رقم مهمة غير صالح.")

while True:
    print("\nاختر خياراً:")
    print("1. إضافة مهمة")
    print("2. عرض المهام")
    print("3. إكمال مهمة")
    print("4. حذف مهمة")
    print("5. خروج")

    choice = input("أدخل اختيارك: ")

    if choice == "1":
        task_name = input("أدخل اسم المهمة: ")
        add_task(task_name)
    elif choice == "2":
        view_tasks()
    elif choice == "3":
        view_tasks()
        try:
            task_num = int(input("أدخل رقم المهمة لإكمالها: "))
            complete_task(task_num - 1) # -1 لأن القوائم تبدأ من 0
```

```
        except ValueError:
            print("مدخل غير صالح. يرجى إدخال رقم")
    elif choice == "4":
        view_tasks()
        try:
            task_num = int(input("أدخل رقم المهمة لحذفها: "))
            delete_task(task_num - 1) # لأن القوائم تبدأ من 0 -1
        except ValueError:
            print("مدخل غير صالح. يرجى إدخال رقم")
    elif choice == "5":
        print("شكراً لاستخدامك برنامج إدارة المهام. إلى اللقاء!")
        break
    else:
        print("اختيار غير صالح. يرجى المحاولة مرة أخرى")
```

الخاتمة

تهانينا! لقد وصلت إلى نهاية هذا الكتاب الشامل حول بايثون للمبتدئين. نأمل أن تكون هذه الرحلة قد زودتك بأساس قوي في لغة البرمجة بايثون، وأن تكون قد اكتسبت الثقة والمهارات اللازمة لمواصلة التعلم والاستكشاف في هذا المجال المثير.

لقد قمنا بتغطية مجموعة واسعة من المواضيع، بدءًا من الأساسيات مثل المتغيرات وهياكل التحكم، مرورًا بالدوال والبرمجة الكائنية التوجه، وصولاً إلى التعامل مع الملفات واستخدام المكتبات الشائعة، وانتهاءً ببناء مشاريع تطبيقية بسيطة. كل مفهوم تعلمته هو لبنة أساسية في بناء قدراتك البرمجية.

تذكر أن تعلم البرمجة هو عملية مستمرة. لا تتوقف عن الممارسة والتجريب. أفضل طريقة لتعزيز مهاراتك هي من خلال بناء المشاريع، حتى لو كانت صغيرة في البداية. كل مشروع جديد سيعلمك شيئًا جديدًا ويواجهك بتحديات ستساعدك على النمو.

الخطوات التالية لتعلم بايثون

الآن بعد أن أصبحت لديك أساسيات بايثون، إليك بعض المجالات التي يمكنك استكشافها لتعميق معرفتك:

- **تطوير الويب:** تعلم أطر عمل الويب مثل Django أو Flask لبناء تطبيقات ويب قوية.
- **علم البيانات والذكاء الاصطناعي:** تعمق في مكتبات مثل Pandas, NumPy, Matplotlib, Scikit-learn, TensorFlow, و PyTorch لتحليل البيانات وبناء نماذج التعلم الآلي.
- **الأتمتة والبرمجة النصية:** استخدم بايثون لأتمتة المهام المتكررة على جهاز الكمبيوتر الخاص بك أو في الشبكات.
- **تطوير الألعاب:** استكشف مكتبات مثل Pygame لإنشاء ألعاب بسيطة.

- **تطوير تطبيقات سطح المكتب:** استخدم مكتبات مثل PyQt أو Tkinter لبناء واجهات مستخدم رسومية.
- **البرمجة التنافسية:** حل المشكلات الخوارزمية لتعزيز مهاراتك في حل المشكلات والتفكير المنطقي.

مصادر إضافية

لا تتردد في البحث عن مصادر إضافية لتعلم بايثون. الإنترنت مليء بالدروس، والوثائق، والمنتديات، والمجتمعات التي يمكن أن تدعم رحلتك التعليمية:

- **الوثائق الرسمية لبايثون:** docs.python.org
- **Real Python:** موقع ممتاز يقدم دروسًا ومقالات متعمقة حول بايثون.
- **freeCodeCamp:** يقدم دورات مجانية وشاملة في البرمجة، بما في ذلك بايثون.
- **Stack Overflow:** منتدى للمبرمجين حيث يمكنك طرح الأسئلة والحصول على إجابات.
- **GitHub:** استكشف مشاريع بايثون مفتوحة المصدر لترى كيف يبني المطورون الآخرون تطبيقاتهم.

نتمنى لك كل التوفيق في رحلتك البرمجية. تذكر، السماء هي الحدود لما يمكنك تحقيقه باستخدام بايثون!

الفهرس

قائمة التمارين والحلول

(سيتم تجميع جميع التمارين والحلول هنا في النسخة النهائية لسهولة الرجوع إليها.)

توسيع الفصل الأول: الأساسيات

المتغيرات وأنواع البيانات (تفصيل إضافي)

الأعداد (Numbers):

- **الأعداد الصحيحة (`int` - Integers):** تُستخدم لتمثيل الأعداد الكاملة الموجبة والسالبة والصفر. لا يوجد حد لحجم العدد الصحيح في بايثون، فهو يتكيف مع حجم الذاكرة المتاحة. `python large_number = 12345678901234567890 print(large_number) print(type(large_number))`
- **الأعداد العشرية (`float` - Floats):** تُستخدم لتمثيل الأعداد الحقيقية التي تحتوي على جزء عشري. يتم تمثيلها عادةً باستخدام الفاصلة العائمة مزدوجة الدقة (double-precision floating-point numbers). `python pi_value = 3.14159 negative_float = -0.001 print(pi_value) print(negative_float) print(type(pi_value))` **ملاحظة:** قد تحدث أخطاء صغيرة في الدقة عند التعامل مع الأعداد العشرية بسبب طريقة تمثيلها في الذاكرة. هذا ليس خاصًا ببايثون بل هو سمة عامة للحوسبة ذات الفاصلة العائمة.

السلاسل النصية (`str` - Strings):

السلاسل النصية هي تسلسلات غير قابلة للتغيير (immutable) من الأحرف. يمكن تعريفها باستخدام علامات اقتباس مفردة (`'`)، مزدوجة (`

`" )، أو ثلاثية (`'''` أو `"""`).

- **علامات الاقتباس الثلاثية:** تُستخدم لإنشاء سلاسل نصية متعددة الأسطر (multiline strings) أو لتوثيق الدوال والفئات (docstrings). `python multiline_string = """هذا سطر أول. وهذا سطر ثانٍ. وأخيرًا، هذا سطر ثالث. """ print(multiline_string)`
- **عمليات السلاسل النصية:**
  - **الدمج (Concatenation):** استخدام علامة `+` لدمج سلسلتين نصيتين. `python first_name = "محمد" last_name = "علي" full_name = first_name + " " + last_name print(full_name)`
  - **التكرار (Repetition):** استخدام علامة `*` لتكرار سلسلة نصية عددًا معينًا من المرات. `python repeated_text = "Hello " * 3 print(repeated_text)`

◦ **الوصول إلى الأحرف (Indexing):** يمكن الوصول إلى أحرف فردية في السلسلة باستخدام الفهارس (indices). الفهرس الأول هو 0. `python my_string = "Python" print(my_string[0]) # P print(my_string[2]) # t print(my_string[-1]) # n (آخر حرف)`

◦ **التقطيع (Slicing):** استخراج جزء (شريحة) من السلسلة النصية. `python my_string = "Programming" print(my_string[0:4]) # Prog print(my_string[4:]) # ramming print(my_string[:3]) # Pro print(my_string[::2]) # Pormig (كل حرفين) print(my_string[::-1]) # gnimmargorP (عكس السلسلة)`

◦ **دوال السلاسل النصية (String Methods):** تحتوي السلاسل النصية على العديد من الدوال المدمجة المفيدة. `python text = " Hello, World! " print(text.strip()) # "Hello, World!" (إزالة المسافات البيضاء من البداية والنهاية) print(text.lower()) # " hello, world! " (تحويل إلى أحرف صغيرة) print(text.upper()) # " HELLO, WORLD! " (تحويل إلى أحرف كبيرة) print(text.replace("World", "Python")) # " Hello, Python! " (استبدال جزء من السلسلة) print("Python is fun".split()) # ["Python", "is", "fun"] (تقسيم السلسلة إلى قائمة) print(", ".join(["apple", "banana", "cherry"])) # apple, banana, cherry (دمج عناصر قائمة بسلسلة) print("Hello".startswith("He")) # True print("World".endswith("ld")) # True print("example.com".find("com")) # 8 (إرجاع فهرس أول ظهور) print("example.com".count("e")) # 2 (عدد مرات ظهور حرف)`

القيم المنطقية (Booleans - `bool`):

تُستخدم لتمثيل قيم الصواب والخطأ. تُعد أساسًا لاتخاذ القرارات في البرمجة.

- `True`: تمثل الصواب.
- `False`: تمثل الخطأ.

`python is_active = True is_admin = False print(is_active and is_admin) # False print(is_active or is_admin) # True print(not is_active) # False`

القوائم (Lists - `list`):

القوائم هي هياكل بيانات قابلة للتغيير (mutable) ومرتبة، يمكن أن تحتوي على عناصر من أنواع بيانات مختلفة. يمكن تعديلها بعد إنشائها.

- **إنشاء القوائم:** `python empty_list = [] numbers = [1, 2, 3, 4, 5] mixed_data = ["apple", 10, True, 3.14]`
- **الوصول إلى العناصر:** باستخدام الفهارس (indexing) والتقطيع (slicing) كما في السلاسل النصية. `python my_list = [10, 20, 30, 40, 50] print(my_list[0]) # 10 print(my_list[2:5]) # [30, 40, 50]`
- **تعديل العناصر:** `python my_list[1] = 25 print(my_list) # [10, 25, 30, 40, 50]`

القواميس (`dict` - Dictionaries):

القواميس هي هياكل بيانات غير مرتبة (في الإصدارات القديمة من بايثون، ولكنها مرتبة منذ بايثون 3.7+) وقابلة للتغيير، تخزن البيانات في أزواج مفتاح-قيمة. المفاتيح يجب أن تكون فريدة وغير قابلة للتغيير.

- **إنشاء القواميس:** `python empty_dict = {} person = {"name": "أحمد", "age": 30, "city": "القاهرة"}`
- **الوصول إلى القيم:** باستخدام المفاتيح. `python print(person["name"]) # أحمد`
- **إضافة/تعديل أزواج المفتاح-القيمة:** `python person["email"] = "ahmed@example.com" person["age"] = 31 print(person)`

المجموعات (`set` - Sets):

المجموعات هي هياكل بيانات غير مرتبة وغير مفهرسة، تحتوي على عناصر فريدة فقط. تُستخدم بشكل أساسي لإزالة التكرارات وإجراء عمليات المجموعات الرياضية.

- **إنشاء المجموعات:** `python my_set = {1, 2, 3, 2, 1} print(my_set) # {1, 2, 3}`

الصفوف (`tuple` - Tuples):

الصفوف هي هياكل بيانات مرتبة وغير قابلة للتغيير (immutable). بمجرد إنشائها، لا يمكن تغيير عناصرها. تُستخدم عادة لتخزين مجموعة من العناصر التي لا يُقصد تغييرها.

- **إنشاء الصفوف:** `python my_tuple = (1, 2, 3) single_element_tuple = (5,) # يجب وضع فاصلة لتمييزه عن تعبير رياضي`
- **الوصول إلى العناصر:** باستخدام الفهارس والتقطيع. `python print(my_tuple[0]) # 1`

العمليات الحسابية والمنطقية (تفصيل إضافي)

ترتيب العمليات (Operator Precedence):

عند وجود عدة عمليات في تعبير واحد، تتبع بايثون ترتيبًا محددًا لتنفيذها، مشابهًا لقواعد الرياضيات (PEMDAS/BODMAS):

1. الأقواس `()`
2. الأس `**`
3. الضرب `*`، القسمة `/`، القسمة الصحيحة `//`، باقي القسمة `%`
4. الجمع `+`، الطرح `-`
5. عمليات المقارنة `==`, `!=`, `<`, `>`, `<=`, `>=`
6. العمليات المنطقية `or`, `and`, `not`

```
result = 10 + 5 * 2 # 10 + 10 = 20
print(result)

result = (10 + 5) * 2 # 15 * 2 = 30
print(result)

result = 2 ** 3 + 1 # 8 + 1 = 9
print(result)

# مثال مع عمليات منطقية
age = 25
has_license = True
is_student = False

if age > 18 and has_license or is_student:
    print("مؤهل لشيء ما.")
else:
    print("غير مؤهل.")
# هنا، `and` لها أولوية أعلى من `or`، لذا يتم تقييم `age > 18 and has_license` أولاً.
# (True and True) or False -> True or False -> True
```

الإدخال والإخراج (تفصيل إضافي)

تنسيق السلاسل النصية (String Formatting):

هناك عدة طرق لتنسيق السلاسل النصية في بايثون لدمج المتغيرات والقيم:

1. **استخدام علامة `%` (قديمة):** `name = "أحمد" age = 30 print("الاسم: %s، العمر: %d" % (name, age))`
2. **استخدام دالة `str.format()`:** `name = "سارة" age = 25 print("الاسم: {}، العمر: {}".format(name, age)) print("الاسم: {0}، العمر: {1}. {0}`

`العمر: ,{n} :الاسم")print (format(name, age)."البرمجة. تحب (format(n=name, a=age)."{a}`

3. **استخدام f-strings (الطريقة المفضلة في بايثون 3.6+):** تُعد f-strings (formatted string literals) الطريقة الأكثر حداثة ومرونة لتنسيق السلاسل النصية. تبدأ بحرف `f` أو `F` قبل علامة الاقتباس، وتسمح بتضمين تعبيرات بايثون مباشرة داخل الأقواس المتعرجة `{}`.

```python name = "خالد"age = 35 print(f"الاسم: {name}، العمر: {age}")

price = 19.99 quantity = 3 total = price * quantity print(f"سعر المنتج: {price:.2f}، الكمية: {quantity}، الإجمالي: {total:.2f}")

# .2f تعني تنسيق الرقم العشري بمنزلتين بعد الفاصلة

print(f"5 + 3 = {5 + 3}") ```

**التعليقات (تفصيل إضافي)**

تُعد التعليقات جزءًا حيويًا من الكود الجيد. إنها لا تؤثر على تنفيذ البرنامج، ولكنها ضرورية لقابلية القراءة والصيانة. يجب أن تكون التعليقات واضحة وموجزة ومفيدة.

- **متى تستخدم التعليقات؟**
  - لشرح سبب اتخاذ قرار برمجي معين (لماذا فعلت هذا؟).
  - لتوضيح الأجزاء المعقدة من الكود.
  - لشرح الغرض من دالة أو فئة أو متغير.
  - لإضافة ملاحظات مؤقتة (مثل TODOs).
- **متى تتجنب التعليقات؟**
  - عندما يكون الكود واضحًا بذاته (لا تعلق على ما يفعله الكود، بل لماذا يفعله).
  - التعليقات القديمة أو غير الدقيقة.

**أمثلة على التعليقات الجيدة والسيئة:**
```

```python
# سيء: يشرح ما يفعله الكود (واضح من الكود نفسه)
x = x + 1 # زيادة x بمقدار 1

# جيد: يشرح لماذا يتم فعل ذلك
# زيادة العداد لتتبع عدد المحاولات الفاشلة قبل إغلاق الحساب
failed_attempts += 1

# سيء: تعليق طويل ومعقد يمكن استبداله بكود أو اسم متغير أفضل
# هذا المتغير يخزن القيمة التي تم إدخالها بواسطة المستخدم
# والتي تم تحويلها إلى عدد صحيح بعد التحقق من صحتها
user_input_as_int = int(input("أدخل رقماً: "))

# جيد: اسم متغير واضح لا يحتاج إلى تعليق
user_age = int(input("أدخل عمرك: "))
```

:Docstrings

تُستخدم Docstrings لتوثيق الوحدات، والفئات، والدوال. يمكن الوصول إليها في وقت التشغيل باستخدام السمة `__doc__` أو الدالة `help()`.

```python
def calculate_rectangle_area(length, width):
    """
    تحسب مساحة المستطيل.

    المعاملات:
    length (float or int): طول المستطيل.
    width (float or int): عرض المستطيل.

    تُرجع:
    float or int: مساحة المستطيل.
    """
    return length * width

print(calculate_rectangle_area.__doc__)
help(calculate_rectangle_area)
```

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل الأول، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل الثاني: هياكل التحكم

الشروط (Conditional Statements) - تفصيل إضافي

تُعد جمل الشروط حجر الزاوية في أي لغة برمجة، حيث تسمح لبرنامجك باتخاذ قرارات بناءً على قيم المتغيرات أو نتائج العمليات. هذا يمنح برامجك القدرة على التفاعل مع المدخلات والبيئات المختلفة.

1. جملة `if`:

تُستخدم لتنفيذ كتلة من الكود فقط إذا كان الشرط المحدد صحيحًا (`True`). إذا كان الشرط خاطئًا (`False`)، فسيتم تخطي كتلة الكود هذه.

```python
# مثال: التحقق من درجة الطالب
student_score = 85

if student_score >= 50:
    print("تهانينا! لقد نجحت في الاختبار.")

# مثال آخر: التحقق من حالة الطقس
is_sunny = True
if is_sunny:
    print("الطقس مشمس، يمكنك الخروج.")
```

2. جملة `if-else`:

توفر مسارًا بديلًا للتنفيذ. إذا كان الشرط في `if` صحيحًا، يتم تنفيذ كتلة `if`. وإلا (أي إذا كان الشرط خاطئًا)، يتم تنفيذ كتلة `else`.

```python
# مثال: التحقق من العمر للتصويت
age = 17

if age >= 18:
    print("أنت مؤهل للتصويت.")
else:
    print("أنت غير مؤهل للتصويت بعد.")

# مثال آخر: التحقق من وجود عنصر في قائمة
items = ["تفاح", "برتقال", "موز"]
search_item = "عنب"

if search_item in items:
    print(f"{search_item} موجود في القائمة.")
else:
    print(f"{search_item} غير موجود في القائمة.")
```

3. جملة `if-elif-else`:

تُستخدم عندما يكون لديك أكثر من شرطين للتحقق منهما. يتم تقييم الشروط بالتسلسل من الأعلى إلى الأسفل. بمجرد العثور على شرط صحيح، يتم تنفيذ كتلة الكود المرتبطة به، ويتم تخطي باقي الشروط وكتلة `else`.

```
# مثال: تحديد تقدير الطالب بناءً على الدرجة
final_score = 92

if final_score >= 90:
    print("تقديرك: ممتاز (A)")
elif final_score >= 80:
    print("تقديرك: جيد جداً (B)")
elif final_score >= 70:
    print("تقديرك: جيد (C)")
elif final_score >= 60:
    print("تقديرك: مقبول (D)")
else:
    print("تقديرك: راسب (F)")

# مثال آخر: تحديد وقت اليوم
hour = 14 # الساعة 2 ظهراً

if hour < 12:
    print("صباح الخير!")
elif hour < 18:
    print("مساء الخير!")
else:
    print("ليلة سعيدة!")
```

الشروط المتداخلة (Nested Conditionals):

يمكنك وضع جمل شرطية داخل جمل شرطية أخرى. هذا مفيد عندما تحتاج إلى التحقق من شروط متعددة بطريقة هرمية.

```python
user_status = "admin"
is_logged_in = True

if is_logged_in:
    print("المستخدم مسجل الدخول.")
    if user_status == "admin":
        print("أهلاً بك أيها المدير، لديك صلاحيات كاملة.")
    elif user_status == "editor":
        print("أهلاً بك أيها المحرر، يمكنك تعديل المحتوى.")
    else:
        print("أهلاً بك أيها المستخدم العادي.")
else:
    print("يرجى تسجيل الدخول أولاً.")
```

الحلقات التكرارية (Loops) - تفصيل إضافي

تُستخدم الحلقات التكرارية لأتمتة المهام المتكررة. بدلاً من كتابة نفس الكود عدة مرات، يمكنك استخدام حلقة لتنفيذه تلقائيًا لعدد معين من المرات أو حتى يتم استيفاء شرط معين.

1. حلقة `for`:

تُستخدم حلقة `for` للتكرار على عناصر تسلسل (مثل القوائم، الصفوف، السلاسل النصية، المجموعات، والقواميس) أو أي كائن قابل للتكرار (iterable).

- **التكرار على قائمة من الأرقام:** `python numbers = [1, 2, 3, 4, 5] for num in numbers: print(f"الرقم: {num}")`
- **التكرار على سلسلة نصية (حرفًا بحرف):** `python word = "Python" for char in word: print(f"الحرف: {char}")`
- **التكرار على قاموس:** عند التكرار على قاموس باستخدام حلقة `for` مباشرة، يتم التكرار على المفاتيح افتراضيًا. ```python student_grades = {"علي": 90, "سارة": 85, "أحمد": 92} for name in student_grades: print(f"اسم الطالب: {name}")

للتكرار على القيم

for grade in student_grades.values(): print(f"الدرجة: {grade}")

للتكرار على المفاتيح والقيم معاً باستخدام ()items

for name, grade in student_grades.items(): print(f"الطالب {name} حصل على درجة {grade}") ```

- **استخدام `range()` مع حلقة `for`:** دالة `range()` مفيدة جدًا لتوليد تسلسلات من الأرقام، وهي شائعة الاستخدام مع حلقات `for` عندما تحتاج إلى التكرار لعدد محدد من المرات.
  - `range(stop)`: تولد أرقامًا من 0 حتى `stop-1`.
  - `range(start, stop)`: تولد أرقامًا من `start` حتى `stop-1`.
  - `range(start, stop, step)`: تولد أرقامًا من `start` حتى `stop-1` بزيادة `step`.

```python

# طباعة الأرقام من 0 إلى 4

for i in range(5): print(i)

# طباعة الأرقام الزوجية من 0 إلى 10

for i in range(0, 11, 2): print(i)

# طباعة الأرقام تنازلياً

for i in range(10, 0, -1): print(i) ```

**2. حلقة `while`:**

تُستخدم حلقة `while` لتنفيذ كتلة من الكود بشكل متكرر طالما أن الشرط المحدد صحيح. بمجرد أن يصبح الشرط خاطئًا، تتوقف الحلقة.
```

```python
# مثال: عد تنازلي
countdown = 5
while countdown > 0:
    print(countdown)
    countdown -= 1 # يجب تحديث المتغير الذي يتحكم في الشرط
print("انطلاق!")

# مثال: طلب مدخلات من المستخدم حتى يتم إدخال قيمة صحيحة
user_input = ""
while not user_input.isdigit(): # طالما المدخل ليس رقماً
    user_input = input("أدخل رقماً صحيحاً: ")
print(f"لقد أدخلت الرقم: {user_input}")
```

الحلقات المتداخلة (Nested Loops):

يمكنك وضع حلقة داخل حلقة أخرى. تُستخدم الحلقات المتداخلة غالبًا للعمل مع هياكل البيانات ثنائية الأبعاد (مثل المصفوفات أو الجداول) أو لإنشاء أنماط معينة.

```python
# مثال: طباعة جدول الضرب
for i in range(1, 6):
    for j in range(1, 6):
        print(f"{i} * {j} = {i * j}\t", end="") # \t لإضافة مسافة أفقية
    print() # لإنشاء سطر جديد بعد كل صف
```

`pass` ,`continue` ,`break` - تفصيل إضافي

تُعد هذه الكلمات المفتاحية أدوات قوية للتحكم الدقيق في تدفق الحلقات.

1. `break`:

تُستخدم لإنهاء الحلقة الحالية فورًا. يتم تخطي أي كود متبقي في الحلقة بعد `break`، ويستمر تنفيذ البرنامج من أول سطر بعد الحلقة.

```python
# مثال: البحث عن عنصر في قائمة
numbers = [10, 20, 30, 40, 50]
search_value = 30

for num in numbers:
    if num == search_value:
        print(f"تم العثور على {search_value}!")
        break # إنهاء الحلقة بمجرد العثور على العنصر
    print(f"جاري البحث... {num}")
else:
    # تُنفذ كتلة else فقط إذا لم يتم الوصول إلى break (أي إذا لم يتم العثور على العنصر)
    print(f"{search_value} غير موجود في القائمة.")
```

2. `continue`:

تُستخدم لتخطي التكرار الحالي للحلقة والانتقال مباشرة إلى التكرار التالي. يتم تجاهل أي كود متبقي في كتلة الحلقة بعد `continue` في التكرار الحالي.

```python
# مثال: طباعة الأرقام الفردية فقط
for i in range(1, 11):
    if i % 2 == 0: # إذا كان الرقم زوجياً
        continue   # تخطى هذا التكرار وانتقل إلى التالي
    print(i)
# الناتج: 1, 3, 5, 7, 9
```

3. `pass`:

هي عملية لا تفعل شيئًا. تُستخدم كعنصر نائب (placeholder) عندما تتطلب بنية الكود جملة، ولكنك لا تريد تنفيذ أي كود في الوقت الحالي. غالبًا ما تُستخدم في المراحل الأولى من تطوير الكود لتجنب أخطاء بناء الجملة.

```
# مثال: دالة لم يتم تنفيذها بعد
def future_feature():
    pass # سيتم إضافة الكود هنا لاحقاً

# مثال: في جملة شرطية
status = "pending"
if status == "completed":
    # do something
    pass
elif status == "pending":
    pass # لا تفعل شيئاً حالياً
else:
    # handle other statuses
    pass
```

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل الثاني، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل الثالث: الدوال

تعريف الدوال واستدعاؤها - تفصيل إضافي

الدوال هي كتل برمجية منظمة وقابلة لإعادة الاستخدام تؤدي مهمة واحدة أو مجموعة من المهام ذات الصلة. استخدام الدوال يجعل الكود أكثر تنظيمًا، وقابلية للقراءة، وسهولة في الصيانة، ويقلل من تكرار الكود (DRY - Don't Repeat Yourself).

بنية تعريف الدالة:

```
def function_name(parameter1, parameter2, ...):
    """Docstring: وصف موجز لما تفعله الدالة."""
    # كود الدالة هنا
    # يمكن أن تتضمن عمليات حسابية، شروط، حلقات، إلخ.
    return result # اختياري: لإرجاع قيمة
```

- `def` : الكلمة المفتاحية التي تبدأ تعريف الدالة.
- `function_name` : اسم الدالة. يجب أن يكون وصفيًا ويعكس وظيفة الدالة.
- `parameters` : (اختياري) المتغيرات التي تستقبلها الدالة كمدخلات. تُعرف داخل الأقواس.
- `:` : نقطتان رأسيتان تشيران إلى بداية كتلة الكود الخاصة بالدالة.
- `Docstring` : سلسلة نصية (عادة محاطة بثلاث علامات اقتباس) توفر وصفًا للدالة. تُعد ممارسة جيدة للتوثيق.

- `return` : (اختياري) الكلمة المفتاحية التي تُستخدم لإرجاع قيمة من الدالة. إذا لم تُرجع الدالة أي شيء صراحة، فإنها تُرجع `None` افتراضيًا.

مثال على دالة بسيطة:

```python
def say_hello():
    """تطبع رسالة ترحيب بسيطة."""
    print("مرحباً بك في عالم الدوال!")

# استدعاء الدالة
say_hello()
```

المعاملات (Parameters) والقيم المعادة (Return Values) - تفصيل إضافي

أنواع المعاملات:

1. **المعاملات الموضعية (Positional Arguments):** تُمرر القيم إلى الدالة بترتيبها. يجب أن يتطابق عدد المعاملات الممررة مع عدد المعاملات المحددة في تعريف الدالة. ```python def describe_person(name, age, city): print(f"الاسم: {name}, العمر: {age}, المدينة: {city}")

describe_person("سارة", 28, "دبي") # الترتيب مهم ```

2. **المعاملات ذات الكلمات المفتاحية (Keyword Arguments):** تُمرر القيم إلى الدالة باستخدام اسم المعامل، مما يجعل الترتيب غير مهم ويزيد من وضوح الكود. `python describe_person(city="دبي", age=28, name="سارة") # الترتيب لا يهم`

3. **المعاملات الافتراضية (Default Parameters):** يمكنك تعيين قيمة افتراضية لمعامل. إذا لم يتم تمرير قيمة لهذا المعامل عند استدعاء الدالة، فسيتم استخدام القيمة الافتراضية. ```python def greet_user(name="زائر"): print(f"أهلاً بك يا {name}!")

greet_user() # أهلاً بك يا زائر! greet_user("فهد") # أهلاً بك يا فهد! ``` **ملاحظة هامة:** يجب أن تأتي المعاملات الافتراضية بعد المعاملات غير الافتراضية في تعريف الدالة.

4. **معاملات الطول المتغير (`*args` و `**kwargs`):**

   - `*args` (arguments): تُستخدم لتمرير عدد متغير من المعاملات الموضعية إلى الدالة. تُجمع هذه المعاملات في `tuple`. ```python def sum_all_numbers(*args): total = 0 for num in args: total += num return total

print(sum_all_numbers(1, 2, 3)) # 6 print(sum_all_numbers(10, 20, 30, 40)) # 100 ` * `**kwargs` (keyword arguments): تُستخدم لتمرير عدد متغير من المعاملات ذات الكلمات المفتاحية. تُجمع هذه المعاملات في`

`dictionary`.python def print_info(**kwargs): for key, value in` kwargs.items(): print(f"{key}: {value}")

print_info(name="ليلى", age=29, city="الرياض") ```

القيم المعادة (Return Values):

تُستخدم الكلمة المفتاحية `return` لإرسال قيمة (أو قيم) من الدالة إلى الجزء الذي استدعاها. يمكن للدالة أن تُرجع أي نوع من البيانات.

- **إرجاع قيمة واحدة:** ```python def get_square(number): return number * number

sq = get_square(7) print(sq) # 49 `* **إرجاع قيم متعددة:** يمكن للدالة أن تُرجع قيمًا متعددة كـ `tuple`. يمكنك بعد ذلك فك حزمة (unpack) هذا الصف إلى متغيرات منفصلة.`python def get_min_max(numbers): return min(numbers), max(numbers)

data = [10, 5, 20, 15, 8] minimum, maximum = get_min_max(data) print(f"الحد الأدنى: {minimum}, الحد الأقصى: {maximum}") ```

الدوال المدمجة (Built-in Functions) - تفصيل إضافي

بايثون توفر مجموعة غنية من الدوال المدمجة التي تُسرع عملية التطوير وتوفر وظائف شائعة. بالإضافة إلى تلك المذكورة سابقًا، إليك بعض الدوال المدمجة الأخرى المفيدة:

- `len()`: تُرجع طول (عدد العناصر) لكائن (سلسلة نصية، قائمة، قاموس، إلخ). `python print(len("Python")) # 6 print(len([1, 2, 3, 4])) # 4`
- `type()`: تُرجع نوع الكائن. `python print(type(10)) # <class 'int'> print(type("hello")) # <class 'str'>`
- `sum()`: تُرجع مجموع العناصر في كائن قابل للتكرار (يجب أن تكون العناصر أرقامًا). `python print(sum([1, 2, 3, 4, 5])) # 15`
- `sorted()`: تُرجع قائمة جديدة مرتبة من عناصر كائن قابل للتكرار. لا تُعدل الكائن الأصلي. `python numbers = [3, 1, 4, 1, 5, 9] sorted_numbers = sorted(numbers) print(sorted_numbers) # [1, 1, 3, 4, 5, 9] print(numbers) # [3, 1, 4, 1, 5, 9] (لم تتغير)`
- `enumerate()`: تُرجع كائنًا من أزواج (فهرس، قيمة) لكائن قابل للتكرار. مفيد عند التكرار على قائمة والحاجة إلى الفهرس والقيمة معًا. `python fruits = ["تفاح", "برتقال", "موز"] for index, fruit in enumerate(fruits): print(f"الفهرس: {index}, الفاكهة: {fruit}")`

- `zip()`: تُستخدم لدمج عناصر من كائنات قابلة للتكرار متعددة في أزواج. `python names = ["علي", "سارة", "أحمد"] ages = [25, 30, 22] for name, age in zip(names, ages): print(f"{name} عمره {age} سنة.")`

نطاق المتغيرات (Variable Scope) - تفصيل إضافي

فهم نطاق المتغيرات أمر بالغ الأهمية لتجنب الأخطاء وضمان أن الكود الخاص بك يعمل كما هو متوقع. النطاق يحدد أين يمكن الوصول إلى المتغير في برنامجك.

قاعدة LEGB (Local, Enclosing, Global, Built-in):

عندما تحاول بايثون الوصول إلى متغير، فإنها تبحث عنه بالترتيب التالي:

1. **Local (L):** داخل الدالة الحالية.
2. **Enclosing (E):** في الدوال المتداخلة (nested functions) من الخارج إلى الداخل.
3. **Global (G):** في المستوى الأعلى للملف (خارج أي دالة).
4. **Built-in (B):** في الدوال والأسماء المدمجة في بايثون (مثل `len`, `print`).

مثال على النطاق المحلي والعام:

```
global_var = "أنا متغير عام"

def my_function():
    local_var = "أنا متغير محلي"
    print(global_var) # يمكن الوصول إلى المتغير العام
    print(local_var)  # يمكن الوصول إلى المتغير المحلي

my_function()
print(global_var)
# print(local_var) # NameError: local_var is not defined (خارج النطاق)
```

الدوال المتداخلة (Nested Functions) ونطاق `Enclosing`:

```python
def outer_function():
    x = 10 # متغير في النطاق المحيط (Enclosing scope)

    def inner_function():
        y = 5 # متغير محلي لـ inner_function
        print(x) # يمكن لـ inner_function الوصول إلى x من النطاق المحيط
        print(y)

    inner_function()
    # print(y) # NameError: y is not defined (خارج نطاق inner_function)

outer_function()
```

الكلمة المفتاحية `nonlocal`:

تُستخدم `nonlocal` لتعديل متغير في النطاق المحيط (enclosing scope) من داخل دالة متداخلة، دون أن يصبح هذا المتغير عامًا.

```python
def outer_function():
    count = 0

    def inner_function():
        nonlocal count # نعلن أننا نريد تعديل count في النطاق المحيط
        count += 1
        print("العداد الداخلي:", count)

    inner_function()
    inner_function()
    print("العداد الخارجي:", count)

outer_function()
```

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل الثالث، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل الرابع: هياكل البيانات المتقدمة

القوائم (Lists) - عمليات متقدمة (تفصيل إضافي)

القوائم هي واحدة من أكثر هياكل البيانات استخدامًا في بايثون نظرًا لمرونتها وقدرتها على تخزين أنواع مختلفة من البيانات. كونها قابلة للتغيير (mutable) يعني أنه يمكن تعديلها بعد إنشائها.

1. إنشاء القوائم:

يمكن إنشاء القوائم باستخدام الأقواس المربعة `[]` أو باستخدام الدالة `list()` .

```
 empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed_data = ["apple", 10, True, 3.14]
list_from_string = list("hello") # ["h", "e", "l", "l", "o"]
list_from_tuple = list((1, 2, 3)) # [1, 2, 3]
print(empty_list, numbers, mixed_data, list_from_string, list_from_tuple)
```

2. الوصول إلى العناصر (Indexing and Slicing):

- **الفهرسة (Indexing):** يمكن الوصول إلى عناصر القائمة باستخدام فهارسها. الفهرس الأول هو 0. يمكن استخدام الفهارس السالبة للوصول من نهاية القائمة (-1 هو العنصر الأخير). `python my_list = [10, 20, 30, 40, 50] print(my_list[0]) # 10 print(my_list[2]) # 30 print(my_list[-1]) # 50 print(my_list[-3]) # 30`
- **التقطيع (Slicing):** استخراج جزء (شريحة) من القائمة. الصيغة هي `[start:end:step]` . `python my_list = [10, 20, 30, 40, 50, 60, 70] print(my_list[1:4]) # [20, 30, 40] (من الفهرس 1 حتى قبل الفهرس 4) print(my_list[::2]) # [10, 30, 50, 70] (كل عنصرين) print(my_list[4:]) # [50, 60, 70] (من الفهرس 4 حتى النهاية) print(my_list[:3]) # [10, 20, 30] (من البداية حتى قبل الفهرس 3) print(my_list[::-1]) # [70, 60, 50, 40, 30, 20, 10] (عكس القائمة)`

3. تعديل العناصر:

يمكن تعديل عناصر القائمة مباشرة باستخدام الفهرسة.

```
 my_list = [10, 20, 30]
my_list[1] = 25 # تعديل العنصر في الفهرس 1
print(my_list) # [10, 25, 30]

# يمكن تعديل شريحة من القائمة
my_list[0:2] = [5, 15]
print(my_list) # [5, 15, 30]
```

4. إضافة عناصر (تفصيل إضافي):

- `append(item)` : تُضيف عنصرًا واحدًا إلى نهاية القائمة. هذه العملية فعالة جدًا. `python fruits = ["تفاح", "برتقال"] fruits.append("موز") print(fruits) # ["تفاح", "برتقال", "موز"]`

- `extend(iterable)`: تُضيف جميع عناصر كائن قابل للتكرار (مثل قائمة أخرى، صف، مجموعة) إلى نهاية القائمة الحالية. تُعد مكافئة لاستخدام `+=` مع قائمة. ```python list1 = [1, 2] list2 = [3, 4] list1.extend(list2) print(list1) # [1, 2, 3, 4]

list1.extend("abc") # تُضيف كل حرف كعنصر منفصل print(list1) # [1, 2, 3, 4, "a", "b", "c"] ``` * `insert(index, item)`: تُضيف عنصرًا في موقع محدد داخل القائمة. هذه العملية قد تكون مكلفة إذا كان الفهرس في بداية القائمة، حيث تتطلب إزاحة جميع العناصر اللاحقة. ```python numbers = [1, 2, 4] numbers.insert(2, 3) # إضافة 3 في الفهرس 2 print(numbers) # [1, 2, 3, 4] ```

5. إزالة عناصر (تفصيل إضافي):

- `remove(item)`: تُزيل أول ظهور للعنصر المحدد من القائمة. إذا لم يكن العنصر موجودًا، تُرفع `ValueError`. `python my_list = [1, 2, 3, 2, 4] my_list.remove(2) print(my_list) # [1, 3, 2, 4] # my_list.remove(5) # ValueError: list.remove(x): x not in list`

- `pop(index)`: تُزيل العنصر في الموقع المحدد وتُرجعه. إذا لم يتم تحديد فهرس، تُزيل وتُرجع آخر عنصر. تُرفع `IndexError` إذا كان الفهرس خارج النطاق. ```python my_list = [10, 20, 30, 40] popped_value = my_list.pop(1) # تُزيل 20 وتُرجعها print(popped_value) # 20 print(my_list) # [10, 30, 40]

last_value = my_list.pop() # تُزيل 40 وتُرجعها print(last_value) # 40 print(my_list) # [10, 30] ``` * `clear()`: تُزيل جميع العناصر من القائمة، مما يجعلها فارغة. `python my_list = [1, 2, 3] my_list.clear() print(my_list) # []` * `del statement`: تُستخدم لحذف عناصر من القائمة باستخدام الفهرس أو الشريحة، أو حتى حذف القائمة بأكملها. ```python my_list = [10, 20, 30, 40, 50] del my_list[2] # حذف العنصر في الفهرس 2 (30) print(my_list) # [10, 20, 40, 50]

del my_list[0:2] # حذف العناصر من الفهرس 0 حتى قبل 2 (10, 20) print(my_list) # [40, 50]

del my_list # حذف القائمة بأكملها من الذاكرة

```
```

6. عمليات أخرى مفيدة (تفصيل إضافي):

- `index(item, start, end)`: تُرجع فهرس أول ظهور للعنصر المحدد. يمكن تحديد نطاق بحث اختياري. تُرفع `ValueError` إذا لم يتم العثور على العنصر. `python numbers = [1, 5, 2, 5, 3] print(numbers.index(5)) # 1 print(numbers.index(5, 2)) # 3 (البحث يبدأ من الفهرس 2)`
- `count(item)`: تُرجع عدد مرات ظهور العنصر المحدد في القائمة. `python numbers = [1, 2, 2, 3, 2, 4] print(numbers.count(2)) # 3 print(numbers.count(5)) # 0`
- `sort()`: تُرتّب القائمة في مكانها (in-place). تُعدل القائمة الأصلية. افتراضيًا، الترتيب تصاعدي. يمكن استخدام `reverse=True` للترتيب التنازلي. ```python numbers = [3, 1, 4, 1, 5, 9, 2] numbers.sort() print(numbers) # [1, 1, 2, 3, 4, 5, 9]

words = ["banana", "apple", "cherry"] words.sort(reverse=True) print(words) # ["cherry", "banana", "apple"]` * `sorted(iterable)`: تُرجع قائمة جديدة مرتبة من عناصر كائن قابل للتكرار. لا تُعدل الكائن الأصلي. مفيدة عندما تريد الاحتفاظ بالترتيب الأصلي.`python original_numbers = [3, 1, 4, 1, 5, 9, 2] new_sorted_list = sorted(original_numbers) print(new_sorted_list) # [1, 1, 2, 3, 4, 5, 9] print(original_numbers) # [3, 1, 4, 1, 5, 9, 2]` * `reverse()`: تعكس ترتيب عناصر القائمة في مكانها. تُعدل القائمة الأصلية.`python my_list = [1, 2, 3, 4, 5] my_list.reverse() print(my_list) # [5, 4, 3, 2, 1]` * `copy()`: تُرجع نسخة سطحية (shallow copy) من القائمة. مهمة لتجنب تعديل القائمة الأصلية عن طريق الخطأ عند العمل مع نسخ.`python list_a = [1, 2, 3] list_b = list_a # هذا ليس نسخاً، بل مجرد مرجع لنفس القائمة list_c = list_a.copy() # هذا نسخ حقيقي

list_a.append(4) print(list_a) # [1, 2, 3, 4] print(list_b) # [1, 2, 3, 4] (تأثرت لأنها نفس الكائن) print(list_c) # [1, 2, 3] (لم تتأثر) ```

القواميس (Dictionaries) - عمليات متقدمة (تفصيل إضافي)

القواميس هي هياكل بيانات قوية لتخزين البيانات في أزواج مفتاح-قيمة. تُعد مثالية عندما تحتاج إلى ربط قيم معينة بمفاتيح فريدة للوصول السريع.

1. إنشاء القواميس:

يمكن إنشاء القواميس باستخدام الأقواس المتعرجة `{}` أو باستخدام الدالة `dict()`.

```
empty_dict = {}
student = {"name": "علي", "age": 20, "major": "هندسة"}
# إنشاء من أزواج (مفتاح, قيمة)
student_dict = dict([("name", "علي"), ("age", 20)])
print(empty_dict, student, student_dict)
```

2. الوصول إلى العناصر (تفصيل إضافي):

- `dict[key]`: الطريقة الأكثر شيوعًا للوصول إلى قيمة مرتبطة بمفتاح. إذا لم يكن المفتاح موجودًا، تُرفع `python person` . `KeyError = {"name": "أحمد", "age": 30}` `print(person["name"]) # أحمد # KeyError: # print(person["country"])` `'country'`
- `dict.get(key, default_value)`: طريقة أكثر أمانًا للوصول إلى القيم. إذا لم يكن المفتاح موجودًا، تُرجع `None` (افتراضيًا) أو القيمة الافتراضية التي تحددها. `python person = {"name": "أحمد", "age": 30} print(person.get("name")) # أحمد` `print person.get("country")) # None print(person.get("country", "غير` `محدد")) # غير محدد`

3. إضافة وتعديل عناصر (تفصيل إضافي):

- `dict[key] = value`: إذا كان المفتاح موجودًا، يتم تحديث قيمته. إذا لم يكن موجودًا، يتم إضافة زوج مفتاح-قيمة جديد. ```python student = {"name": "ليلى", "age": 22} student["major"] = "علوم حاسب" # إضافة عنصر جديد # print(student) {"name": "ليلى", "age": 22, "major": "علوم حاسب"}

student["age"] = 23 # تعديل قيمة عنصر موجود # print(student) {"name": "ليلى", "age": 23, "major": "علوم حاسب"} * `update(other_dict)`: تُضيف أزواج مفتاح-قيمة من قاموس آخر إلى القاموس الحالي. إذا كانت هناك مفاتيح مشتركة، يتم تحديث قيمها. python dict1 = {"a": 1, "b": 2} dict2 = {"b": 3, "c": 4} dict1.update(dict2) print(dict1) # {"a": 1, "b": 3, "c": 4} ```

4. إزالة عناصر (تفصيل إضافي):

- `del dict[key]`: تُزيل زوج المفتاح-القيمة المحدد. تُرفع `KeyError` إذا لم يكن المفتاح موجودًا. `python my_dict = {"a": 1, "b": 2, "c": 3} del my_dict["b"]` `print(my_dict) # {"a": 1, "c": 3}`
- `pop(key, default_value)`: تُزيل المفتاح وتُرجع قيمته. إذا لم يكن المفتاح موجودًا، تُرفع `KeyError` ما لم يتم توفير `default_value`. ```python my_dict = {"name": "سارة",

:"name"} # age = my_dict.pop("age") print(age) # 30 print(my_dict) {age": 30" {"سارة"

`*` غير موجود # print(country) ("غير موجود" ,"country")country = my_dict.pop
`popitem()`: تُزيل وتُرجع زوج مفتاح-قيمة عشوائي (في الإصدارات الحديثة من بايثون، تُزيل آخر عنصر تم إدخاله). تُرفع `KeyError` إذا كان القاموس فارغًا. python my_dict = {"a": 1, "b": 2} item = my_dict.popitem() print(item) #
("b", 2) (قد يختلف الترتيب في الإصدارات القديمة) `* `clear` {"a": 1} # print(my_dict) `()`:
تُزيل جميع العناصر من القاموس. python my_dict = {"a": 1, "b": 2} my_dict.clear()
``` {} # print(my_dict)

### 5. طرق عرض القاموس (تفصيل إضافي):

- `keys()`: تُرجع كائن عرض (view object) لجميع المفاتيح في القاموس. هذا الكائن يتغير ديناميكيًا مع تغير القاموس. `python student = {"name": "أحمد", "age": 25} keys = student.keys() print(keys) # dict_keys(["name", "age"]) student["city"] = "جدة" print(keys) # dict_keys(["name", "age", "city"])`
- `values()`: تُرجع كائن عرض لجميع القيم في القاموس. `python values = student.values() print(values) # dict_values(["أحمد", 25, "جدة"])`
- `items()`: تُرجع كائن عرض لجميع أزواج المفتاح-القيمة كصفوف (tuples). `python items = student.items() print(items) # dict_items([("name", "أحمد"), ("age", 25), ("city", "جدة")])`

### المجموعات (Sets) - عمليات متقدمة (تفصيل إضافي)

المجموعات هي هياكل بيانات غير مرتبة وغير مفهرسة، تحتوي على عناصر فريدة فقط. تُستخدم بشكل ممتاز للتحقق من العضوية، وإزالة التكرارات، وإجراء عمليات المجموعات الرياضية.

### 1. إنشاء المجموعات:

- باستخدام الأقواس المتعرجة `{}` (للمجموعات غير الفارغة). إذا استخدمت `{}` لإنشاء مجموعة فارغة، فستنشئ قاموسًا فارغًا. لإنشاء مجموعة فارغة، استخدم `set()`. ```python my_set = {1, 2, 3, 2} print(my_set) # {1, 2, 3}

empty_set = set() print(empty_set) # set()

list_to_set = set([1, 2, 2, 3, 4, 4]) print(list_to_set) # {1, 2, 3, 4} ```

### 2. إضافة وإزالة عناصر (تفصيل إضافي):
```

- `add(item)`: تُضيف عنصرًا واحدًا إلى المجموعة. إذا كان العنصر موجودًا بالفعل، لا يحدث شيء. `python my_set = {1, 2} my_set.add(3) print(my_set) # {1, 2, 3} my_set.add(2) # لا يحدث شيء {1, 2, 3} # print(my_set)`
- `remove(item)`: تُزيل العنصر المحدد. تُرفع `KeyError` إذا لم يكن العنصر موجودًا. `python my_set = {1, 2, 3} my_set.remove(2) print(my_set) # {1, 3} # my_set.remove(5) # KeyError: 5`
- `discard(item)`: تُزيل العنصر المحدد إذا كان موجودًا. إذا لم يكن موجودًا، لا تفعل شيئًا ولا تُرفع أي خطأ. `python my_set = {1, 2, 3} my_set.discard(2) print(my_set) # {1, 3} my_set.discard(5) # لا يحدث شيء {1, 3} # print(my_set)`
- `pop()`: تُزيل وتُرجع عنصرًا عشوائيًا من المجموعة. تُرفع `KeyError` إذا كانت المجموعة فارغة. `python my_set = {1, 2, 3} popped_item = my_set.pop() print(popped_item, my_set) # (1, {2, 3}) (الترتيب عشوائي)`
- `clear()`: تُزيل جميع العناصر من المجموعة. `python my_set = {1, 2, 3} my_set.clear() print(my_set) # set()`

3. عمليات المجموعات الرياضية (تفصيل إضافي):

تُعد المجموعات مفيدة جدًا لإجراء عمليات المجموعات القياسية.

- **الاتحاد (Union):** تُرجع مجموعة جديدة تحتوي على جميع العناصر الفريدة من كلا المجموعتين. يمكن استخدام الدالة `union()` أو العامل `|`. `python A = {1, 2, 3} B = {3, 4, 5} print(A.union(B)) # {1, 2, 3, 4, 5} print(A | B) # {1, 2, 3, 4, 5}`
- **التقاطع (Intersection):** تُرجع مجموعة جديدة تحتوي على العناصر المشتركة بين المجموعتين. يمكن استخدام الدالة `intersection()` أو العامل `&`. `python A = {1, 2, 3} B = {3, 4, 5} print(A.intersection(B)) # {3} print(A & B) # {3}`
- **الفرق (Difference):** تُرجع مجموعة جديدة تحتوي على العناصر الموجودة في المجموعة الأولى وليست في الثانية. يمكن استخدام الدالة `difference()` أو العامل `-`. `python A = {1, 2, 3} B = {3, 4, 5} print(A.difference(B)) # {1, 2} (العناصر في A وليست في B) print(A - B) # {1, 2}`
- **الفرق المتماثل (Symmetric Difference):** تُرجع مجموعة جديدة تحتوي على العناصر الموجودة في إحدى المجموعتين ولكن ليست في كليهما. يمكن استخدام الدالة `symmetric_difference()` أو العامل `^`. `python A = {1, 2, 3} B = {3, 4, 5} print(A.symmetric_difference(B)) # {1, 2, 4, 5} print(A ^ B) # {1, 2, 4, 5}`
- **المجموعات الفرعية والمجموعات الفائقة (Subsets and Supersets):**

- `issubset()`: تُرجع `True` إذا كانت جميع عناصر المجموعة الحالية موجودة في المجموعة الأخرى.
- `issuperset()`: تُرجع `True` إذا كانت جميع عناصر المجموعة الأخرى موجودة في المجموعة الحالية. `python set1 = {1, 2} set2 = {1, 2, 3, 4} print(set1.issubset(set2)) # True print(set2.issuperset(set1)) # True`

الصفوف (Tuples) - تفصيل إضافي

الصفوف هي هياكل بيانات مرتبة وغير قابلة للتغيير (immutable). بمجرد إنشائها، لا يمكن تغيير عناصرها. هذا يجعلها مفيدة لتخزين البيانات التي لا يُقصد تعديلها، مثل الإحداثيات أو سجلات البيانات الثابتة.

1. إنشاء الصفوف:

يمكن إنشاء الصفوف باستخدام الأقواس `()` أو بدونها (تُعد الفواصل هي التي تحدد الصف).

```
 empty_tuple = ()
my_tuple = (1, 2, 3)
single_element_tuple = (5,) # يجب وضع فاصلة لتمييزه عن تعبير رياضي
# إنشاء صف بدون أقواس
coordinates = 10, 20
print(empty_tuple, my_tuple, single_element_tuple, coordinates)
```

2. الوصول إلى العناصر (Indexing and Slicing):

تمامًا مثل القوائم والسلاسل النصية، يمكن الوصول إلى عناصر الصفوف باستخدام الفهرسة والتقطيع.

```
 my_tuple = ("apple", "banana", "cherry", "date")
print(my_tuple[0])    # apple
print(my_tuple[1:3])  # ("banana", "cherry")
print(my_tuple[::-1]) # ("date", "cherry", "banana", "apple")
```

3. عدم قابلية التغيير (Immutability):

هذه هي السمة الرئيسية للصفوف. لا يمكنك إضافة، إزالة، أو تعديل العناصر بعد إنشاء الصف.

```
 my_tuple = (1, 2, 3)
# my_tuple[0] = 5 # TypeError: 'tuple' object does not support item assignment
# my_tuple.append(4) # AttributeError: 'tuple' object has no attribute
'append'
```

4. استخدامات الصفوف:

- **إرجاع قيم متعددة من دالة:** رأينا هذا في الفصل الثالث، حيث تُرجع الدوال قيمًا متعددة كصف.
- **مفاتيح القواميس:** نظرًا لأن الصفوف غير قابلة للتغيير، يمكن استخدامها كمفاتيح في القواميس (على عكس القوائم).
- **حماية البيانات:** عندما تريد التأكد من أن البيانات لا تتغير عن طريق الخطأ.

```
# استخدام صف كمفتاح في قاموس
location = {(34.05, -118.25): "Los Angeles", (40.71, -74.00): "New York"}
print(location[(34.05, -118.25)]) # Los Angeles
```

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل الرابع، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل الخامس: التعامل مع الملفات

يُعد التعامل مع الملفات جزءًا أساسيًا من العديد من التطبيقات البرمجية. تسمح لك بايثون بقراءة البيانات من الملفات وكتابة البيانات إليها بسهولة، مما يتيح لك تخزين المعلومات بشكل دائم ومعالجتها. في هذا الفصل، سنتعمق في كيفية التفاعل مع الملفات النصية.

قراءة وكتابة الملفات النصية - تفصيل إضافي

للتفاعل مع الملفات في بايثون، نستخدم الدالة المدمجة `open()`. تُرجع هذه الدالة كائن ملف (file object) يمكننا استخدامه لإجراء عمليات القراءة أو الكتابة.

1. فتح الملفات (`open()`):

الصيغة الأساسية للدالة `open()` هي:

```
file_object = open(filename, mode, encoding)
```

- `filename` : المسار إلى الملف الذي تريد فتحه (يمكن أن يكون اسم الملف فقط إذا كان في نفس الدليل).
- `mode` : وضع الفتح، يحدد الغرض من فتح الملف. بعض الأوضاع الشائعة:
  - `'r'` (read): الوضع الافتراضي. لفتح الملف للقراءة فقط. إذا لم يكن الملف موجودًا، تُرفع `FileNotFoundError`.
  - `'w'` (write): لفتح الملف للكتابة. إذا كان الملف موجودًا، يتم مسح محتواه. إذا لم يكن موجودًا، يتم إنشاء ملف جديد.

- `'a'` (append): لفتح الملف للكتابة، مع إضافة المحتوى إلى نهاية الملف. إذا لم يكن الملف موجودًا، يتم إنشاء ملف جديد.
- `'x'` (exclusive creation): لإنشاء ملف جديد والكتابة فيه. إذا كان الملف موجودًا بالفعل، تُرفع `FileExistsError`.
- `'b'` (binary): لفتح الملف في الوضع الثنائي (للتعامل مع الصور، الفيديو، إلخ). يُستخدم مع `a` ,`w` ,`r` (مثال: `'wb'`, `'rb'`).
- `'t'` (text): الوضع الافتراضي. لفتح الملف في الوضع النصي (للتعامل مع النصوص). يُستخدم مع `a` ,`w` ,`r` (مثال: `'wt'`, `'rt'`).
- `'+'` (update): لفتح الملف للقراءة والكتابة. يُستخدم مع `a` ,`w` ,`r` (مثال: `'r+'`, `'w+'`, `'a+'`).

- `encoding`: (اختياري) تحديد ترميز الأحرف للملف (مثل `'cp1256'`, `'utf-8'`). يُنصح دائمًا بتحديده لتجنب مشاكل الترميز، خاصة عند التعامل مع اللغات غير الإنجليزية.

2. إغلاق الملفات (`close()`):

من الأهمية بمكان إغلاق الملف بعد الانتهاء من التعامل معه. هذا يحرر الموارد التي يستخدمها الملف ويضمن حفظ أي تغييرات. إذا لم يتم إغلاق الملف، قد تحدث مشاكل مثل فقدان البيانات أو تلف الملف.

```
 file = open("my_file.txt", "w")
file.write("هذا سطر جديد.")
file.close()
```

3. استخدام `with` statement (الطريقة المفضلة):

تُعد جملة `with` الطريقة الأكثر أمانًا وفعالية للتعامل مع الملفات. تضمن هذه الجملة إغلاق الملف تلقائيًا حتى لو حدث خطأ أثناء التعامل معه، مما يجنبك نسيان استدعاء `close()`.

```
 with open("my_file.txt", "w", encoding="utf-8") as file:
    file.write("هذا سطر مكتوب باستخدام with statement.\n")
    file.write("هذا سطر آخر.\n")
# الملف يتم إغلاقه تلقائياً هنا
```

4. الكتابة إلى الملفات:

- `write(string)`: تُستخدم لكتابة سلسلة نصية واحدة إلى الملف. لا تُضيف سطرًا جديدًا تلقائيًا. `python with open("output.txt", "w", encoding="utf-8") as f: f.write("السطر الأول.\n") f.write("السطر الثاني.\n")`

- `writelines(list_of_strings)`: تُستخدم لكتابة قائمة من السلاسل النصية إلى الملف. لا تُضيف أسطرًا جديدة تلقائيًا بعد كل عنصر. `python lines = ["مرحباً بالعالم!\n", "بايثون رائعة.\n", "تعلم ممتع.\n"] with open("lines.txt", "w", encoding="utf-8") as f: f.writelines(lines)`

5. القراءة من الملفات:

- `read(size)`: تُستخدم لقراءة عدد معين من الأحرف من الملف. إذا لم يتم تحديد `size`، تُقرأ جميع محتويات الملف. `python with open("lines.txt", "r", encoding="utf-8") as f: content = f.read() print(content)`
- `readline()`: تُستخدم لقراءة سطر واحد من الملف في كل مرة. تُرجع سلسلة نصية فارغة عند الوصول إلى نهاية الملف. `python with open("lines.txt", "r", encoding="utf-8") as f: line1 = f.readline() line2 = f.readline() print(line1, end="") # end="" لمنع إضافة سطر جديد إضافي print(line2, end="")`
- `readlines()`: تُرجع قائمة تحتوي على جميع الأسطر في الملف. كل سطر هو عنصر في القائمة، ويتضمن حرف السطر الجديد (`\n`). `python with open("lines.txt", "r", encoding="utf-8") as f: all_lines = f.readlines() for line in all_lines: print(line, end="")`
- **التكرار على كائن الملف مباشرة (الطريقة المفضلة لقراءة الأسطر):** يمكنك التكرار مباشرة على كائن الملف لقراءة الأسطر سطرًا سطرًا. هذه الطريقة فعالة من حيث الذاكرة، خاصة للملفات الكبيرة. `python with open("lines.txt", "r", encoding="utf-8") as f: for line in f: print(line, end="")`

6. مؤشر الملف (`tell()` و `seek()`):

- `tell()`: تُرجع الموضع الحالي لمؤشر الملف (عدد البايتات من بداية الملف).
- `seek(offset, whence)`: تُغير موضع مؤشر الملف.
    - `offset`: عدد البايتات للتحرك.
    - `whence`: نقطة البداية (0: بداية الملف، 1: الموضع الحالي، 2: نهاية الملف).

```python with open("sample.txt", "w+", encoding="utf-8") as f: f.write("Hello World!\nPython is great.") print("الموضع الحالي:", f.tell()) # 30 (بعد كتابة النص)
```

```
 f.seek(0) # العودة إلى بداية الملف
print("المحتوى بعد العودة للبداية:", f.read())

f.seek(6) # الانتقال إلى الفهرس 6
print("المحتوى من الفهرس 6:", f.read())
```

```

### التعامل مع الأخطاء (try, except, finally) عند التعامل مع الملفات - تفصيل إضافي

عند التعامل مع الملفات، من الشائع حدوث أخطاء (استثناءات) مثل عدم وجود الملف، أو عدم وجود أذونات للكتابة. يجب عليك دائمًا استخدام كتل `try-except` لمعالجة هذه الأخطاء بشكل رشيق.

**1. `FileNotFoundError`:**

يحدث هذا الاستثناء عندما تحاول فتح ملف للقراءة ( `'r'` ) وهو غير موجود.

```python
 try:
    with open("non_existent_file.txt", "r", encoding="utf-8") as f:
        content = f.read()
        print(content)
except FileNotFoundError:
    print("خطأ: الملف المطلوب غير موجود.")
except Exception as e:
    print(f"حدث خطأ آخر: {e}")
```

2. `PermissionError`:

يحدث هذا الاستثناء عندما لا يكون لديك الأذونات اللازمة لقراءة أو كتابة ملف في موقع معين.

```python
 try:
    # محاولة الكتابة إلى مسار يتطلب أذونات إدارية (مثال على Linux/macOS)
    with open("/root/restricted_file.txt", "w", encoding="utf-8") as f:
        f.write("هذا نص سري.")
except PermissionError:
    print("خطأ: لا تملك الأذونات اللازمة للكتابة في هذا الموقع.")
except Exception as e:
    print(f"حدث خطأ آخر: {e}")
```

3. `IOError` (أو `OSError`):

`IOError` هو اسم قديم لـ `OSError`، ويحدث لأسباب عامة تتعلق بعمليات الإدخال/الإخراج، مثل محاولة الكتابة إلى قرص ممتلئ أو ملف تالف.
```

```python
try:
    # مثال افتراضي لخطأ I/O (قد لا يحدث في بيئة عادية)
    with open("corrupted_disk_file.txt", "w", encoding="utf-8") as f:
        # محاكاة خطأ I/O
        raise IOError("القرص ممتلئ أو تالف!")
except IOError as e:
    print(f"خطأ في الإدخال/الإخراج: {e}")
except Exception as e:
    print(f"حدث خطأ آخر: {e}")
```

4. استخدام `finally` مع الملفات:

على الرغم من أن `with` statement تُغلق الملف تلقائيًا، إلا أن `finally` لا تزال مفيدة لتنفيذ كود تنظيف آخر بغض النظر عما إذا حدث استثناء أم لا.

```python
file_object = None # تهيئة المتغير
try:
    file_object = open("data.txt", "r", encoding="utf-8")
    content = file_object.read()
    print(content)
except FileNotFoundError:
    print("الملف غير موجود.")
except Exception as e:
    print(f"حدث خطأ: {e}")
finally:
    if file_object: # التحقق مما إذا كان كائن الملف قد تم إنشاؤه
        file_object.close()
        print("تم إغلاق الملف في كتلة finally.")
```

أمثلة متقدمة على التعامل مع الملفات:

1. معالجة ملف CSV بسيط:

ملفات CSV (Comma Separated Values) هي تنسيق شائع لتخزين البيانات الجدولية. يمكن التعامل معها كملفات نصية عادية.

```python
# إنشاء ملف CSV تجريبي
csv_data = "Name,Age,City\nAlice,30,New York\nBob,24,London\nCharlie,35,Paris"
with open("people.csv", "w", encoding="utf-8") as f:
    f.write(csv_data)

# قراءة ومعالجة ملف CSV
with open("people.csv", "r", encoding="utf-8") as f:
    header = f.readline().strip().split(",") # قراءة الرأس
    print(f"الرأس: {header}")

    for line in f:
        data = line.strip().split(",")
        person_dict = dict(zip(header, data))
        print(person_dict)
```

2. نسخ ملف:

```python
# إنشاء ملف مصدر
with open("source.txt", "w", encoding="utf-8") as f:
    f.write("هذا هو محتوى الملف المصدر.\n")
    f.write("سيتم نسخه إلى ملف آخر.")

# نسخ المحتوى
try:
    with open("source.txt", "r", encoding="utf-8") as source_file:
        with open("destination.txt", "w", encoding="utf-8") as dest_file:
            for line in source_file:
                dest_file.write(line)
    print("تم نسخ الملف بنجاح.")
except FileNotFoundError:
    print("خطأ: الملف المصدر غير موجود.")
except Exception as e:
    print(f"حدث خطأ أثناء النسخ: {e}")
```

3. إضافة ترقيم للأسطر في ملف:

```python
# إنشاء ملف تجريبي
with open("poem.txt", "w", encoding="utf-8") as f:
    f.write("الليل يا ليلى يعاتبني\n")
    f.write("ويقول لي سلم على ليلى\n")
    f.write("الليل يا ليلى يعاتبني\n")

# إضافة ترقيم للأسطر
try:
    with open("poem.txt", "r", encoding="utf-8") as input_file:
        with open("numbered_poem.txt", "w", encoding="utf-8") as output_file:
            line_number = 1
            for line in input_file:
                output_file.write(f"{line_number}: {line}")
                line_number += 1
    print("تم ترقيم الأسطر بنجاح في numbered_poem.txt")
except FileNotFoundError:
    print("خطأ: ملف القصيدة غير موجود.")
except Exception as e:
    print(f"حدث خطأ أثناء ترقيم الأسطر: {e}")
```

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل الخامس، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل السادس: البرمجة الكائنية التوجه (OOP)

البرمجة الكائنية التوجه (Object-Oriented Programming - OOP) هي نموذج برمجي يعتمد على مفهوم "الكائنات" (Objects)، والتي يمكن أن تحتوي على بيانات (تُسمى خصائص أو سمات - attributes) وكود (تُسمى دوال أو طرق - methods). تُعد OOP طريقة قوية لتنظيم الكود، مما يجعله أكثر قابلية لإعادة الاستخدام، وسهولة في الصيانة، وأكثر وضوحًا في تمثيل الكيانات في العالم الحقيقي.

المفاهيم الأساسية - تفصيل إضافي

1. الفئات (Classes):

الفئة هي مخطط (blueprint) أو قالب لإنشاء الكائنات. تُعرف الفئة الخصائص (البيانات) والسلوكيات (الدوال) التي ستمتلكها الكائنات التي تُنشأ منها.

```python
class Dog:
    # خصائص الفئة (Class attributes) - مشتركة بين جميع الكائنات
    species = "Canis familiaris"

    def __init__(self, name, age):
        # خصائص الكائن (Instance attributes) - خاصة بكل كائن
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} ينبح: هوف هوف!"

    def get_age_in_dog_years(self):
        return self.age * 7
```

- `class Dog:` : تعريف الفئة `Dog`.
- `species` : `species = "Canis familiaris"` هي خاصية فئة. جميع كائنات `Dog` ستشارك نفس قيمة `species`.
- `__init__(self, name, age)` : هذه دالة خاصة تُسمى "المُنشئ" (constructor). تُنفذ تلقائيًا عند إنشاء كائن جديد من الفئة. `self` هو المعامل الأول دائمًا، ويشير إلى الكائن نفسه. `name` و `age` هي خصائص خاصة بكل كائن.
- `self.name = name` : تُعين القيمة الممررة لـ `name` إلى خاصية `name` الخاصة بالكائن.
- `bark(self)` و `get_age_in_dog_years(self)` : هذه هي دوال (أو طرق) الكائن. تُعرف السلوكيات التي يمكن للكائن القيام بها. يجب أن يكون `self` هو المعامل الأول دائمًا.

2. الكائنات (Objects) / النسخ (Instances):

الكائن هو نسخة (instance) من الفئة. عند إنشاء كائن، فإنه يمتلك الخصائص والسلوكيات المحددة في الفئة.

```python
# إنشاء كائنات (نسخ) من الفئة Dog
my_dog = Dog("باستر", 5)
your_dog = Dog("لوسي", 3)

# الوصول إلى خصائص الكائن
print(f"اسم كلبي: {my_dog.name}")
print(f"عمر كلبك: {your_dog.age}")

# استدعاء دوال الكائن
print(my_dog.bark())
print(f"{your_dog.name} عمرها بالسنوات الكلبية: {your_dog.get_age_in_dog_years()} سنة.")

# الوصول إلى خاصية الفئة
print(Dog.species) # Canis familiaris
print(my_dog.species) # Canis familiaris (يمكن الوصول إليها عبر الكائن أيضاً)
```

الوراثة (Inheritance) - تفصيل إضافي

الوراثة هي آلية تسمح لفئة (الفئة الفرعية أو الفئة المشتقة) باكتساب الخصائص والسلوكيات من فئة أخرى (الفئة الأساسية أو الفئة الأم). هذا يعزز قابلية إعادة الاستخدام ويقلل من تكرار الكود.

مثال: لنفترض أن لدينا فئة `Animal`، ثم فئات أكثر تحديدًا مثل `Dog` و `Cat` ترث من `Animal`.

```python
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):
        raise NotImplementedError("يجب على الفئات الفرعية تنفيذ هذه الدالة")

    def eat(self):
        return f"{self.name} يأكل."

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age) # استدعاء مُنشئ الفئة الأم
        self.breed = breed

    def speak(self):
        return f"{self.name} ينبح: هوف هوف!"

    def fetch(self):
        return f"{self.name} يجلب الكرة."

class Cat(Animal):
    def __init__(self, name, age, color):
        super().__init__(name, age)
        self.color = color

    def speak(self):
        return f"{self.name} يمواء: مواء مواء!"

    def scratch(self):
        return f"{self.name} يخدش الأثاث."

# إنشاء كائنات من الفئات الفرعية
my_dog = Dog("باستر", 5, "جولدن ريتريفر")
my_cat = Cat("مياو", 2, "أبيض")

print(my_dog.name) # باستر
print(my_dog.age)  # 5
print(my_dog.breed) # جولدن ريتريفر
print(my_dog.speak()) # باستر ينبح: هوف هوف!
print(my_dog.eat())   # باستر يأكل.
print(my_dog.fetch()) # باستر يجلب الكرة.

print(my_cat.name) # مياو
print(my_cat.color) # أبيض
print(my_cat.speak()) # مياو يمواء: مواء مواء!
print(my_cat.scratch()) # مياو يخدش الأثاث.
```

- `class Dog(Animal)`: تعني أن `Dog` ترث من `Animal`.
- `super().__init__(name, age)`: تُستخدم لاستدعاء مُنشئ الفئة الأم. هذا يضمن تهيئة الخصائص الموروثة من الفئة الأم.
- **تجاوز الطرق (Method Overriding):** الفئة الفرعية يمكنها توفير تنفيذ خاص بها لطريقة موجودة بالفعل في الفئة الأم (مثل `speak()` في هذا المثال). هذا يسمح بتخصيص السلوكيات الموروثة.

التغليف (Encapsulation) - تفصيل إضافي

التغليف هو مبدأ في OOP يهدف إلى تجميع البيانات (الخصائص) والدوال (الطرق) التي تعمل على تلك البيانات في وحدة واحدة (الفئة)، وإخفاء التفاصيل الداخلية لعمل الكائن عن العالم الخارجي. هذا يحمي البيانات من التعديل غير المقصود ويجعل الكود أكثر تنظيمًا.

في بايثون، لا يوجد مفهوم صارم للوصول الخاص (private access) كما في بعض اللغات الأخرى (مثل Java أو C++). ومع ذلك، هناك اصطلاحات تُستخدم للإشارة إلى أن الخاصية أو الطريقة يجب أن تُعامل على أنها خاصة.

- **الخصائص العامة (Public Attributes):** يمكن الوصول إليها وتعديلها مباشرة من خارج الفئة. هذا هو الافتراضي في بايثون. ```python class Car: def **init**(self, make, model): self.make = make self.model = model

  my_car = Car("Toyota", "Camry") print(my_car.make) # Toyota my_car.model = "Corolla" # يمكن تعديلها مباشرة print(my_car.model) ```

- **الخصائص المحمية (Protected Attributes):** تُشير إليها بادئة واحدة من الشرطة السفلية (`_`). تُعد اصطلاحًا للمطورين بأن هذه الخاصية أو الطريقة مخصصة للاستخدام الداخلي للفئة أو الفئات الفرعية، ولا ينبغي الوصول إليها مباشرة من الخارج. ```python class BankAccount: def **init**(self, balance): self._balance = balance # خاصية محمية

```
 def get_balance(self):
    return self._balance

def deposit(self, amount):
    if amount > 0:
        self._balance += amount
        print(f"تم إيداع {amount}. الرصيد الجديد: {self._balance}")
    else:
        print("مبلغ الإيداع يجب أن يكون موجباً.")
```

account = BankAccount(1000) print(account.get_balance()) # 1000

print(account._balance) # يمكن الوصول إليها، لكن لا يُنصح بذلك

``` account.deposit(500)

- **الخصائص الخاصة (Private Attributes):** تُشير إليها بادئتان من الشرطة السفلية (`__`). تُسبب بايثون "تغيير الاسم" (name mangling) لهذه الخصائص، مما يجعل الوصول إليها من الخارج أكثر صعوبة (ولكن ليس مستحيلاً). الهدف هو منع الوصول المباشر وتجنب تضارب الأسماء في الوراثة. ```python class Person: def **init**(self, name, age): self.__name = name # خاصية خاصة self.__age = age

```
 def get_name(self):
    return self.__name

def get_age(self):
    return self.__age
```

p = Person("أحمد", 30) print(p.get_name()) # أحمد

# print(p.__name) # AttributeError: 'Person' object has no attribute '__name'

# print(p._Person__name) # يمكن الوصول إليها بهذه الطريقة، لكن لا يُنصح بها

```

المُعدّلات (Getters) والمُعيّنات (Setters):

للوصول إلى الخصائص المحمية أو الخاصة وتعديلها بطريقة منظمة، غالبًا ما تُستخدم دوال خاصة تُسمى المُعدّلات (Getters) للحصول على القيمة، والمُعيّنات (Setters) لتعيين القيمة. هذا يسمح لك بإضافة منطق تحقق (validation) عند تعديل البيانات.

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    def get_celsius(self):
        return self._celsius

    def set_celsius(self, value):
        if value < -273.15: # الصفر المطلق
            print("درجة الحرارة لا يمكن أن تكون أقل من الصفر المطلق.")
        else:
            self._celsius = value

    def get_fahrenheit(self):
        return (self._celsius * 9/5) + 32

    def set_fahrenheit(self, value):
        self._celsius = (value - 32) * 5/9

temp = Temperature(25)
print(f"درجة الحرارة بالسلسيوس: {temp.get_celsius()}")
print(f"درجة الحرارة بالفهرنهايت: {temp.get_fahrenheit()}")

temp.set_celsius(30)
print(f"درجة الحرارة الجديدة بالسلسيوس: {temp.get_celsius()}")

temp.set_celsius(-300) # محاولة تعيين قيمة غير صالحة
```

التجريد (Abstraction) - تفصيل إضافي

التجريد هو مبدأ في OOP يركز على إظهار المعلومات الأساسية فقط وإخفاء التفاصيل المعقدة وغير الضرورية عن المستخدم. الهدف هو تبسيط واجهة التفاعل مع الكائن، مما يجعلها أسهل في الاستخدام والفهم.

في بايثون، يمكن تحقيق التجريد بعدة طرق، بما في ذلك استخدام الفئات المجردة (Abstract Classes) والطرق المجردة (Abstract Methods).

الفئات المجردة والطرق المجردة:

الفئة المجردة هي فئة لا يمكن إنشاء كائنات منها مباشرة. يجب أن ترث منها فئات فرعية، وهذه الفئات الفرعية يجب أن توفر تنفيذًا للطرق المجردة المحددة في الفئة الأم.

لإنشاء فئات مجردة في بايثون، نستخدم وحدة (Abstract Base Classes) `abc`.

```
from abc import ABC, abstractmethod

class Shape(ABC): # فئة مجردة
    @abstractmethod
    def area(self):
        pass # طريقة مجردة

    @abstractmethod
    def perimeter(self):
        pass # طريقة مجردة

    def describe(self):
        return "هذه فئة مجردة للأشكال الهندسية."

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# لا يمكن إنشاء كائن من فئة مجردة
# s = Shape() # TypeError: Can't instantiate abstract class Shape with
abstract methods area, perimeter

circle = Circle(5)
print(f"مساحة الدائرة: {circle.area()}")
print(f"محيط الدائرة: {circle.perimeter()}")
print(circle.describe())

rectangle = Rectangle(4, 6)
print(f"مساحة المستطيل: {rectangle.area()}")
print(f"محيط المستطيل: {rectangle.perimeter()}")
```

- `from abc import ABC, abstractmethod` : استيراد الوحدات اللازمة.
- `class Shape(ABC):` : الإشارة إلى أن `Shape` هي فئة مجردة عن طريق الوراثة من `ABC` .
- `@abstractmethod` : مُزخرف (decorator) يُستخدم لتعريف طريقة مجردة. الفئات الفرعية التي ترث من `Shape` يجب أن توفر تنفيذًا لهذه الطرق.

التجريد يساعد في تصميم أنظمة معقدة عن طريق التركيز على "ماذا" تفعل الكائنات بدلاً من "كيف" تفعلها، مما يؤدي إلى كود أكثر نظافة وسهولة في الإدارة.

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل السادس، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل السابع: الوحدات والحزم (Modules and Packages)

تُعد الوحدات والحزم من المفاهيم الأساسية في بايثون التي تُمكنك من تنظيم الكود الخاص بك بطريقة منطقية وفعالة. مع نمو المشاريع، يصبح من الضروري تقسيم الكود إلى أجزاء أصغر وأكثر قابلية للإدارة. هذا لا يحسن فقط من قابلية القراءة والصيانة، بل يعزز أيضًا قابلية إعادة استخدام الكود.

1. الوحدات (Modules) - تفصيل إضافي

الوحدة في بايثون هي ببساطة ملف يحتوي على كود بايثون. يمكن أن يحتوي هذا الملف على تعريفات للدوال، والفئات، والمتغيرات، وحتى كود قابل للتنفيذ. عندما تستورد وحدة، فإنك تجعل جميع التعريفات الموجودة فيها متاحة للاستخدام في ملفك الحالي.

أمثلة على الوحدات المدمجة (Built-in Modules):

بايثون تأتي مع مكتبة قياسية ضخمة تحتوي على العديد من الوحدات المدمجة التي توفر وظائف لمجموعة واسعة من المهام. بعض الأمثلة الشائعة:

- `math` : يوفر دوال رياضية متقدمة (مثل الجذر التربيعي، اللوغاريتمات، الدوال المثلثية). `python import math print(math.sqrt(16)) # 4.0 print(math.pi) # 3.141592653589793`
- `random` : يوفر دوال لتوليد أرقام عشوائية. `python import random print(random.randint(1, 10)) # رقم عشوائي بين 1 و 10 (شامل) print(random.choice(["تفاح", "برتقال", "موز"])) # اختيار عشوائي من قائمة`
- `datetime` : يوفر فئات للتعامل مع التواريخ والأوقات. `python import datetime now = datetime.datetime.now() print(f"التاريخ والوقت الحالي: {now}") print(f"السنة: {now.year}, الشهر: {now.month}, اليوم: {now.day}")`

- `os`: يوفر واجهة للتفاعل مع نظام التشغيل (مثل التعامل مع الملفات والدلائل). `python import os # الحصول على دليل العمل الحالي print(f"دليل العمل الحالي: {os.getcwd()}") # التحقق مما إذا كان ملف موجوداً print(os.path.exists("my_file.txt"))`

طرق الاستيراد (تفصيل إضافي):

- **`import module_name`**: هذه هي الطريقة الأكثر شيوعًا. تستورد الوحدة بأكملها، ويجب عليك استخدام اسم الوحدة كبادئة للوصول إلى محتوياتها. `python import my_module result = my_module.my_function()`
- **`import module_name as alias`**: تُستخدم لتعيين اسم مستعار (alias) للوحدة، مما يجعل الكود أقصر وأسهل في القراءة، خاصة للوحدات ذات الأسماء الطويلة. `python import numpy as np arr = np.array([1, 2, 3])`
- **`from module_name import object_name`**: تستورد أسماء محددة (دوال، فئات، متغيرات) مباشرة من الوحدة. يمكنك استخدام هذه الأسماء مباشرة دون الحاجة إلى بادئة اسم الوحدة. `python from math import sqrt, pi print(sqrt(25)) print(pi)`
- **`from module_name import *`**: تستورد جميع الأسماء من الوحدة. على الرغم من أنها تبدو مريحة، إلا أنها لا تُنصح بها في البرامج الكبيرة لأنها قد تؤدي إلى تضارب في الأسماء (name clashes) إذا كانت هناك أسماء متطابقة في وحدات مختلفة. `python from math import * print(sqrt(49)) print(pi)`

متى يتم تنفيذ كود الوحدة؟

عندما يتم استيراد وحدة لأول مرة، يتم تنفيذ الكود الموجود فيها من الأعلى إلى الأسفل. إذا تم استيراد نفس الوحدة مرة أخرى، فلن يتم تنفيذ الكود مرة أخرى (بايثون تقوم بتخزين الوحدات المستوردة مؤقتًا).

غالبًا ما ترى الكود التالي في ملفات الوحدات:

```
# my_module.py
def main():
    print("هذا الكود يُنفذ عند تشغيل الملف مباشرة.")

if __name__ == "__main__":
    main()
```

- `if __name__ == "__main__":`: هذا الشرط يتحقق مما إذا كان الملف الحالي هو البرنامج الرئيسي الذي يتم تشغيله مباشرة، وليس وحدة يتم استيرادها. الكود داخل هذه الكتلة سيُنفذ فقط عندما يتم تشغيل الملف كبرنامج رئيسي.

2. إنشاء الوحدات الخاصة بك - تفصيل إضافي

إنشاء وحداتك الخاصة أمر بسيط للغاية. كل ما عليك فعله هو حفظ الكود الخاص بك في ملف بايثون (`.py`). ثم يمكنك استيراده في ملفات بايثون أخرى في نفس الدليل أو في دليل يمكن لبايثون العثور عليه.

مثال عملي:

لنقم بإنشاء وحدة بسيطة لإدارة قائمة المهام.

ملف `todo_module.py`:

```
# todo_module.py
tasks = []

def add_task(task_name):
    tasks.append({"name": task_name, "completed": False})
    print(f"تم إضافة المهمة: {task_name}")

def view_tasks():
    if not tasks:
        print("لا توجد مهام حالياً.")
        return
    print("\nقائمة المهام:")
    for i, task in enumerate(tasks):
        status = "[مكتمل]" if task["completed"] else "[غير مكتمل]"
        print(f"{i + 1}. {task["name"]} {status}")

def complete_task(task_index):
    if 0 <= task_index < len(tasks):
        tasks[task_index]["completed"] = True
        print(f"تم وضع علامة على المهمة \'{tasks[task_index]["name"]}\' كمكتملة.")
    else:
        print("رقم مهمة غير صالح.")

if __name__ == "__main__":
    print("تشغيل وحدة المهام مباشرة.")
    add_task("شراء البقالة")
    add_task("دراسة بايثون")
    view_tasks()
    complete_task(0)
    view_tasks()
```

ملف `main_app.py` (في نفس الدليل)

```python
# main_app.py
import todo_module

print("تشغيل التطبيق الرئيسي.")

todo_module.add_task("إعداد العشاء")
todo_module.add_task("قراءة كتاب")
todo_module.view_tasks()
todo_module.complete_task(0)
todo_module.view_tasks()

# لاحظ أن قائمة المهام في todo_module هي نفسها التي يتم التعديل عليها
# لأن الوحدة يتم استيرادها مرة واحدة فقط.
```

3. الحزم (Packages) - تفصيل إضافي

الحزمة هي طريقة لتنظيم الوحدات ذات الصلة في بنية دليل هرمية. تُعد الحزم مفيدة بشكل خاص للمشاريع الكبيرة التي تحتوي على العديد من الوحدات التي تحتاج إلى تنظيم منطقي. الحزمة هي في الأساس مجلد يحتوي على ملف `__init__.py` (يمكن أن يكون فارغًا) وملفات وحدات أخرى أو حزم فرعية.

الغرض من `__init__.py`:

- يُشير إلى أن الدليل الذي يحتويه هو حزمة بايثون.
- يُنفذ عند استيراد الحزمة أو أي وحدة داخلها لأول مرة.
- يمكن استخدامه لتهيئة الحزمة، أو تحديد المتغيرات التي ستكون متاحة عند استيراد الحزمة مباشرة، أو لتحديد الوحدات التي سيتم استيرادها تلقائيًا عند استخدام `from package import *`.

مثال على هيكل حزمة معقد:

```
my_project/
├── main.py
└── my_package/
    ├── __init__.py
    ├── data_processing/
    │   ├── __init__.py
    │   ├── cleaning.py
    │   └── analysis.py
    └── visualization/
        ├── __init__.py
        ├── charts.py
        └── maps.py
```

أمثلة على الاستيراد من حزمة:

لنفرض أن `cleaning.py` يحتوي على دالة `clean_text`، و `charts.py` يحتوي على دالة `generate_bar_chart`.

```
# main.py

# استيراد وحدة محددة من حزمة فرعية
from my_package.data_processing import cleaning
from my_package.visualization import charts

text = "  Hello World!  "
cleaned_text = cleaning.clean_text(text)
print(f"النص المنظف: {cleaned_text}")

data = [10, 20, 15]
labels = ["A", "B", "C"]
charts.generate_bar_chart(data, labels)

# استيراد دالة محددة مباشرة
from my_package.data_processing.cleaning import clean_text
print(clean_text("  Another Text  "))

# استيراد الحزمة الفرعية نفسها
import my_package.data_processing
# الآن يمكنك الوصول إلى الوحدات داخلها
# my_package.data_processing.analysis.some_function()
```

4. إدارة الحزم (pip) - تفصيل إضافي

`pip` هو مدير الحزم الرسمي لبايثون. يُستخدم لتثبيت وإدارة الحزم الخارجية (المكتبات) التي لا تأتي مدمجة مع بايثون. هذه الحزم تُنشر عادةً على PyPI (Python Package Index)، وهو مستودع مركزي لحزم بايثون.

أوامر `pip` الشائعة (تفصيل إضافي):

- **`pip install package_name`**: لتثبيت حزمة واحدة أو أكثر. `bash pip install requests beautifulsoup4`
- **`pip install package_name==version_number`**: لتثبيت إصدار محدد من حزمة. هذا مفيد لضمان توافق المشروع. `bash pip install pandas==1.3.5`
- **`pip install --upgrade package_name`**: لترقية حزمة مثبتة إلى أحدث إصدار متاح. `bash pip install --upgrade numpy`

- **`pip uninstall package_name`**: لإلغاء تثبيت حزمة. `bash pip uninstall matplotlib`
- **`pip list`**: لعرض جميع الحزم المثبتة في البيئة الحالية مع أرقام إصداراتها. `bash pip list`
- **`pip show package_name`**: لعرض معلومات مفصلة حول حزمة معينة، مثل الإصدار، الموقع، التبعيات، والمؤلف. `bash pip show requests`
- **`pip freeze > requirements.txt`**: لإنشاء ملف `requirements.txt` يحتوي على قائمة بجميع الحزم المثبتة في البيئة الحالية مع أرقام إصداراتها الدقيقة. هذا أمر بالغ الأهمية لإعادة إنتاج بيئة المشروع. `bash pip freeze > requirements.txt`
- **`pip install -r requirements.txt`**: لتثبيت جميع الحزم المدرجة في ملف `requirements.txt` . هذا يسمح للمطورين الآخرين أو لأنظمتك الآلية بإعداد بيئة المشروع بسهولة. `bash pip install -r requirements.txt`

البيئات الافتراضية (Virtual Environments):

تُعد البيئات الافتراضية ممارسة جيدة جدًا في تطوير بايثون. تسمح لك بإنشاء بيئات بايثون معزولة لكل مشروع. هذا يمنع تضارب التبعيات بين المشاريع المختلفة.

- **إنشاء بيئة افتراضية:** `bash python3 -m venv my_project_env`
- **تنشيط البيئة الافتراضية:**
  - على Linux/macOS: `bash source my_project_env/bin/activate`
  - على Windows (Command Prompt): `bash my_project_env\Scripts\activate.bat`
  - على Windows (PowerShell): `bash my_project_env\Scripts\Activate.ps1`
- **إلغاء تنشيط البيئة الافتراضية:** `bash deactivate`

عندما تكون البيئة الافتراضية نشطة، فإن أي حزم تقوم بتثبيتها باستخدام `pip` ستُثبت داخل تلك البيئة فقط، ولن تؤثر على تثبيت بايثون الرئيسي أو المشاريع الأخرى.

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل السابع، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل الثامن: التعامل مع الأخطاء والاستثناءات

في أي برنامج، بغض النظر عن مدى بساطته أو تعقيده، من المحتم أن تحدث أخطاء. هذه الأخطاء يمكن أن تكون نتيجة لأخطاء في الكود نفسه (أخطاء برمجية)، أو بسبب مدخلات غير صحيحة من المستخدم، أو مشاكل في البيئة (مثل عدم وجود ملف). تُعد القدرة على التعامل مع هذه الأخطاء بشكل رشيق أمرًا بالغ الأهمية لإنشاء برامج قوية وموثوقة لا تنهار فجأة.

1. أنواع الأخطاء الشائعة - تفصيل إضافي

فهم الأنواع المختلفة من الأخطاء يساعدك على تشخيص المشكلات وحلها بفعالية.

- **أخطاء بناء الجملة (Syntax Errors):** تحدث هذه الأخطاء عندما لا يتبع الكود قواعد بناء الجملة للغة بايثون. يكتشفها مفسر بايثون (interpreter) قبل أن يبدأ في تنفيذ الكود. إذا كان هناك خطأ في بناء الجملة، فلن يتم تشغيل البرنامج على الإطلاق. غالبًا ما تُشير رسالة الخطأ إلى السطر الذي حدث فيه الخطأ ونوع الخطأ.

```python

# مثال على خطأ بناء الجملة: قوس مفقود

---

print("مرحباً بالعالم" # SyntaxError: unexpected EOF while parsing

---

# مثال آخر: كلمة مفتاحية مكتوبة بشكل خاطئ

---

if x == 10

---

print("X is 10") # SyntaxError: expected ":"

---
```

- **الاستثناءات (Exceptions) / أخطاء وقت التشغيل (Runtime Errors):** تحدث الاستثناءات أثناء تنفيذ البرنامج (بعد أن يتم فحص بناء الجملة بنجاح). تُشير إلى أن شيئًا غير متوقع قد حدث، مما يمنع البرنامج من الاستمرار في التنفيذ بشكل طبيعي. إذا لم يتم التعامل مع الاستثناء، فإنه سيؤدي إلى توقف البرنامج (crash).

أمثلة شائعة على الاستثناءات:

  - `NameError` : تُرفع عندما تحاول استخدام متغير أو دالة لم يتم تعريفها. `python # print(undefined_variable) # NameError: name 'undefined_variable' is not defined`
  - `TypeError` : تُرفع عندما يتم تطبيق عملية أو دالة على كائن من نوع غير مناسب. `python # "hello" + 5 # TypeError: can only concatenate str (not "int") to str # len(123) # TypeError: object of type 'int' has no len()`
  - `ValueError` : تُرفع عندما تكون العملية صحيحة من حيث النوع، ولكن القيمة غير مناسبة. `python # int("abc") # ValueError: invalid literal for int() with base 10: 'abc'`
  - `ZeroDivisionError` : تُرفع عند محاولة القسمة على صفر. `python # 10 / 0 # ZeroDivisionError: division by zero`
  - `IndexError` : تُرفع عند محاولة الوصول إلى فهرس خارج نطاق قائمة أو صف أو سلسلة نصية. `python # my_list = [1, 2, 3] # print(my_list[3]) # IndexError: list index out of range`
  - `KeyError` : تُرفع عند محاولة الوصول إلى مفتاح غير موجود في قاموس. `python # my_dict = {"a": 1} # print(my_dict["b"]) # KeyError: 'b'`
  - `FileNotFoundError` : تُرفع عند محاولة فتح ملف للقراءة وهو غير موجود. `python # open("non_existent.txt", "r") # FileNotFoundError: [Errno 2] No such file or directory: 'non_existent.txt'`
  - `AttributeError` : تُرفع عندما تحاول الوصول إلى خاصية أو دالة غير موجودة لكائن. `python # my_string = "hello" # my_string.append("world") # AttributeError: 'str' object has no attribute 'append'`

2. معالجة الاستثناءات (Exception Handling) - تفصيل إضافي

تُعد معالجة الاستثناءات هي الآلية التي تسمح لك بالتعامل مع أخطاء وقت التشغيل بشكل رشيق، مما يمنع برنامجك من الانهيار. يتم ذلك باستخدام كتل `else` ، `except` ، `try` و `finally` .

أ. كتلة `try-except` :

- `try`: تضع الكود الذي تتوقع أن يرفع استثناءً بداخله. إذا حدث استثناء في هذه الكتلة، يتم إيقاف تنفيذ الكود داخل `try` ، ويتم البحث عن كتلة `except` مطابقة.
- `except`: تُحدد نوع الاستثناء الذي تريد التقاطه. إذا حدث الاستثناء المحدد في كتلة `try` ، يتم تنفيذ الكود داخل كتلة `except` .

```python def safe_divide(numerator, denominator): try: result = numerator / denominator print(f"النتيجة: {result}") except ZeroDivisionError: print("خطأ: لا يمكن القسمة على صفر.") except TypeError: print("خطأ: يجب أن تكون المدخلات أرقاماً.") except Exception as e: # التقاط أي استثناء آخر وتخزين رسالته في المتغير e print(f"حدث خطأ غير متوقع: {e}")

safe_divide(10, 2) # النتيجة: 5.0 safe_divide(10, 0) # خطأ: لا يمكن القسمة على صفر. safe_divide(10, "a") # خطأ: يجب أن تكون المدخلات أرقاماً. safe_divide("x", 2) # حدث خطأ غير متوقع: unsupported operand type(s) for /: 'str' and 'int' ```

**ملاحظات على `except`:** * يمكنك تحديد عدة كتل `except` لأنواع مختلفة من الاستثناءات. سيتم تنفيذ أول كتلة `except` تطابق الاستثناء الذي حدث. * يمكنك التقاط عدة استثناءات في كتلة `except` واحدة عن طريق وضعها في `tuple`. `python try: value = int(input("أدخل رقماً: ")) result = 10 / value print(f"النتيجة: {result}") except (ValueError, ZeroDivisionError): print("خطأ: يرجى إدخال رقم صحيح غير صفري.")` * إذا لم تحدد نوع الاستثناء بعد `except` ، فإنه سيلتقط أي نوع من الاستثناءات. هذه الممارسة لا تُنصح بها عمومًا لأنها قد تخفي أخطاء برمجية حقيقية.

### ب. كتلة `else`:

تُنفذ كتلة `else` فقط إذا لم تحدث أي استثناءات في كتلة `try` . تُستخدم لوضع الكود الذي يجب أن يُنفذ فقط عند نجاح عملية `try` .

```python
try:
    file_name = input("أدخل اسم الملف للقراءة: ")
    with open(file_name, "r", encoding="utf-8") as f:
        content = f.read()
except FileNotFoundError:
    print(f"خطأ: الملف \'{file_name}\' غير موجود.")
else:
    print("تم قراءة الملف بنجاح. المحتوى:")
    print(content)
```

ج. كتلة `finally`:
```

تُنفذ كتلة `finally` دائمًا، بغض النظر عما إذا حدث استثناء أم لا، أو ما إذا تم التعامل معه. تُستخدم عادة لتنظيف الموارد، مثل إغلاق الملفات، أو اتصالات الشبكة، أو تحرير الذاكرة، لضمان أن هذه العمليات تتم دائمًا.

```
def process_file(path):
    file = None
    try:
        file = open(path, "r", encoding="utf-8")
        content = file.read()
        print("محتوى الملف:\n", content)
    except FileNotFoundError:
        print(f"خطأ: الملف \'{path}\' غير موجود.")
    except Exception as e:
        print(f"حدث خطأ غير متوقع: {e}")
    finally:
        if file: # التحقق مما إذا كان كائن الملف قد تم إنشاؤه بنجاح
            file.close()
            print("تم إغلاق الملف في كتلة finally.")

process_file("existing_file.txt") # افترض وجود هذا الملف
process_file("non_existent_file.txt")
```

3. رفع الاستثناءات (Raising Exceptions) - تفصيل إضافي

بالإضافة إلى التقاط الاستثناءات، يمكنك أيضًا رفع استثناءات خاصة بك (أو استثناءات بايثون المدمجة) باستخدام الكلمة المفتاحية `raise`. هذا مفيد عندما تريد فرض شروط معينة في الكود الخاص بك، أو عندما تكتشف حالة خطأ لا يمكن للدالة التعامل معها.

```
def set_age(age):
    if not isinstance(age, (int, float)):
        raise TypeError("العمر يجب أن يكون رقماً.")
    if age < 0 or age > 120:
        raise ValueError("العمر يجب أن يكون بين 0 و 120.")
    print(f"تم تعيين العمر إلى: {age}")

try:
    set_age(30)
    set_age(-5) # هنا سيُرفع ValueError
    set_age("abc") # قبله ValueError لن يتم الوصول إلى هذا السطر إذا تم رفع
except ValueError as ve:
    print(f"خطأ في القيمة: {ve}")
except TypeError as te:
    print(f"خطأ في النوع: {te}")
```

الاستثناءات المخصصة (Custom Exceptions):

يمكنك تعريف استثناءات خاصة بك عن طريق إنشاء فئة جديدة ترث من فئة `Exception` المدمجة أو أي من فئاتها الفرعية. هذا يجعل الكود أكثر وضوحًا وقابلية للقراءة، حيث يمكنك إنشاء استثناءات تعكس الأخطاء الخاصة بمنطق عمل تطبيقك.

```python
class InvalidInputError(Exception):
    """استثناء يُرفع عندما يكون المدخل غير صالح."""
    def __init__(self, message="مدخل غير صالح.", value=None):
        self.message = message
        self.value = value
        super().__init__(self.message)

class InsufficientFundsError(Exception):
    """استثناء يُرفع عندما تكون الأموال غير كافية لإجراء عملية."""
    def __init__(self, message="أموال غير كافية.", current_balance=0,
required_amount=0):
        self.message = message
        self.current_balance = current_balance
        self.required_amount = required_amount
        super().__init__(self.message)

def withdraw(balance, amount):
    if not isinstance(amount, (int, float)) or amount <= 0:
        raise InvalidInputError("مبلغ السحب يجب أن يكون رقماً موجباً.", amount)
    if amount > balance:
        raise InsufficientFundsError("رصيد غير كافٍ للسحب.", balance, amount)
    return balance - amount

# اختبار الاستثناءات المخصصة
try:
    current_balance = 1000
    new_balance = withdraw(current_balance, 1500)
    print(f"الرصيد الجديد: {new_balance}")
except InsufficientFundsError as e:
    print(f"خطأ: {e.message} الرصيد الحالي: {e.current_balance}, المبلغ
المطلوب: {e.required_amount}")
except InvalidInputError as e:
    print(f"خطأ: {e.message} القيمة المدخلة: {e.value}")
except Exception as e:
    print(f"حدث خطأ عام: {e}")

try:
    current_balance = 500
    new_balance = withdraw(current_balance, "abc")
except InsufficientFundsError as e:
    print(f"خطأ: {e.message} الرصيد الحالي: {e.current_balance}, المبلغ
المطلوب: {e.required_amount}")
except InvalidInputError as e:
    print(f"خطأ: {e.message} القيمة المدخلة: {e.value}")
except Exception as e:
    print(f"حدث خطأ عام: {e}")
```

أفضل الممارسات في معالجة الاستثناءات:

- **كن محددًا:** التقط الاستثناءات الأكثر تحديدًا أولاً، ثم الاستثناءات العامة. تجنب استخدام `except Exception` بشكل عام ما لم تكن هناك حاجة ماسة لذلك، لأنها قد تخفي أخطاء غير متوقعة.
- **لا تبتلع الأخطاء:** لا تترك كتلة `except` فارغة أو تطبع رسالة عامة جدًا دون معالجة حقيقية. يجب أن تفعل شيئًا مفيدًا (تسجيل الخطأ، إبلاغ المستخدم، محاولة استرداد).
- **استخدم `finally` للتنظيف:** استخدم `finally` لضمان تحرير الموارد (مثل الملفات أو اتصالات قاعدة البيانات) بغض النظر عما يحدث.
- **لا تستخدم الاستثناءات للتحكم في التدفق العادي:** الاستثناءات مخصصة للحالات الاستثنائية، وليس للتحكم في التدفق الطبيعي للبرنامج. على سبيل المثال، لا تستخدم `try-except` للتحقق مما إذا كان عنصر موجودًا في قائمة؛ استخدم `if item in list` بدلاً من ذلك.
- **وثّق الاستثناءات:** إذا كانت دالتك ترفع استثناءات معينة، فقم بتوثيقها في Docstring الخاصة بالدالة.

هذا التوسع يضيف تفاصيل أعمق وأمثلة أكثر لكل قسم في الفصل الثامن، مما يساعد على زيادة المحتوى وتحسين الفهم. سأستمر في توسيع الفصول الأخرى بنفس الطريقة.

توسيع الفصل التاسع: مقدمة إلى المكتبات الشائعة

تُعد بايثون لغة برمجة متعددة الاستخدامات، ويعود جزء كبير من هذه المرونة إلى نظامها البيئي الغني بالمكتبات والحزم. هذه المكتبات توفر وظائف جاهزة ومُحسّنة لمجموعة واسعة من المهام، مما يوفر على المبرمجين عناء كتابة الكود من الصفر. في هذا الفصل، سنتعمق في ثلاث من أهم المكتبات وأكثرها استخدامًا في مجالات الحوسبة العلمية، وتحليل البيانات، وتصور البيانات: NumPy، Pandas، و Matplotlib.

1. NumPy (Numerical Python) - تفصيل إضافي

NumPy هي المكتبة الأساسية للحوسبة العلمية في بايثون. توفر كائن مصفوفة قويًا يسمى `ndarray` (N-dimensional array) ودوال للعمل مع هذه المصفوفات بكفاءة عالية. تُعد NumPy حجر الزاوية للعديد من مكتبات بايثون الأخرى في مجالات علم البيانات والتعلم الآلي.

لماذا NumPy؟

- **السرعة والكفاءة:** عمليات NumPy على المصفوفات أسرع بكثير من العمليات على قوائم بايثون العادية، خاصة مع مجموعات البيانات الكبيرة. هذا يرجع إلى أن NumPy مكتوبة جزئيًا بلغة C، وتستخدم تقنيات تحسين الأداء.
- **استهلاك الذاكرة:** تستهلك مصفوفات NumPy ذاكرة أقل بكثير من قوائم بايثون لنفس الكمية من البيانات.

- **العمليات المتجهة (Vectorized Operations):** تسمح لك بإجراء عمليات رياضية على المصفوفات بأكملها دفعة واحدة دون الحاجة إلى حلقات تكرارية صريحة، مما يجعل الكود أقصر وأكثر قابلية للقراءة.

الميزات الرئيسية لـ NumPy:

- **كائن المصفوفة `ndarray`:** هو كائن مصفوفة متعدد الأبعاد ومتجانس (يحتوي على عناصر من نفس النوع). يمكن أن تكون المصفوفات أحادية البعد (متجهات)، ثنائية البعد (مصفوفات)، أو متعددة الأبعاد.

```python import numpy as np

# إنشاء مصفوفة أحادية البعد (vector)

arr1d = np.array([1, 2, 3, 4, 5]) print("المصفوفة أحادية البعد:", arr1d) print("الشكل (الأبعاد):", arr1d.shape) # (5,) print("نوع البيانات:", arr1d.dtype) # int64 (أو حسب النظام)

# إنشاء مصفوفة ثنائية البعد (matrix)

arr2d = np.array([[1, 2, 3], [4, 5, 6]]) print("المصفوفة ثنائية البعد:\n", arr2d) print("الشكل (الأبعاد):", arr2d.shape) # (2, 3) print("عدد الأبعاد:", arr2d.ndim) # 2

# إنشاء مصفوفة ثلاثية الأبعاد

arr3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]) print("المصفوفة ثلاثية الأبعاد:\n", arr3d) print("الشكل (الأبعاد):", arr3d.shape) # (2, 2, 2) ```

- **إنشاء المصفوفات:**
    - `np.array()`: لإنشاء مصفوفة من قائمة أو صف بايثون.
    - `np.zeros(shape)`: لإنشاء مصفوفة مملوءة بالأصفار.
    - `np.ones(shape)`: لإنشاء مصفوفة مملوءة بالواحدات.
    - `np.empty(shape)`: لإنشاء مصفوفة بدون تهيئة (تحتوي على قيم عشوائية من الذاكرة).
```

- `np.arange(start, stop, step)`: لإنشاء مصفوفة بقيم متسلسلة (مشابهة لـ `range()`).
- `np.linspace(start, stop, num)`: لإنشاء مصفوفة بقيم متساوية التباعد ضمن نطاق محدد.
- `np.random.rand(d0, d1, ...)`: لإنشاء مصفوفة بأرقام عشوائية من توزيع موحد.

`python print("مصفوفة أصفار 2x2:\n", np.zeros((2, 2))) print("مصفوفة واحدات 3x1:\n", np.ones((3, 1))) print("مصفوفة متسلسلة:\n", np.arange(0, 10, 2)) # [0 2 4 6 8] print("مصفوفة متساوية التباعد:\n", np.linspace(0, 1, 5)) # [0. 0.25 0.5 0.75 1. ] print("مصفوفة عشوائية 2x2:\n", np.random.rand(2, 2))`

- **الوصول إلى العناصر والتقطيع (Indexing and Slicing):** الوصول إلى عناصر مصفوفات NumPy يشبه القوائم، ولكن مع إمكانيات أكثر قوة للأبعاد المتعددة.

```python arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) print("المصفوفة الأصلية:\n", arr)

print("العنصر في الصف 0، العمود 1:", arr[0, 1]) # 2 print("الصف الأول:", arr[0]) # [1 2 3] print("العمود الثاني (جميع الصفوف، العمود 1):\n", arr[:, 1]) # [2 5 8] print("شريحة 2x2 من الزاوية العلوية اليسرى:\n", arr[0:2, 0:2]) ```

- **العمليات الرياضية على المصفوفات:** تُطبق العمليات الرياضية الأساسية (الجمع، الطرح، الضرب، القسمة) عنصرًا بعنصر.

```python arr1 = np.array([[1, 2], [3, 4]]) arr2 = np.array([[5, 6], [7, 8]])

print("الجمع:\n", arr1 + arr2) print("الضرب (عنصر بعنصر):\n", arr1 * arr2) print("ضرب المصفوفات (dot product):\n", np.dot(arr1, arr2)) print("الجذر التربيعي:\n", np.sqrt(arr1)) print("مجموع جميع العناصر:", np.sum(arr1)) print("مجموع كل عمود:", np.sum(arr1, axis=0)) # axis=0 يعني على طول الأعمدة (مجموع الصفوف) print("مجموع كل صف:", np.sum(arr1, axis=1)) # axis=1 يعني على طول الصفوف (مجموع الأعمدة) ```

### 2. Pandas - تفصيل إضافي

Pandas هي مكتبة مفتوحة المصدر توفر هياكل بيانات سهلة الاستخدام وعالية الأداء وأدوات لتحليل البيانات ومعالجتها في بايثون. تُعد Pandas أداة لا غنى عنها في علم البيانات، حيث تُستخدم على نطاق واسع لتنظيف البيانات، وتحويلها، وتحليلها، ودمجها.

**الميزات الرئيسية لـ Pandas:**

- `Series`: هيكل بيانات أحادي البعد يشبه المصفوفة أو العمود في جدول بيانات. يمكن أن يحتوي على أي نوع من البيانات.

python import pandas as pd```

``` print("Series:\n", s) يمثل قيمة مفقودة s = pd.Series([1, 3, 5, np.nan, 6, 8]) # np.nan

- `DataFrame`: هيكل بيانات ثنائي الأبعاد، يشبه جدول البيانات في Excel أو قاعدة البيانات العلائقية. يتكون من صفوف وأعمدة، حيث كل عمود هو كائن `Series`.

python```

# إنشاء DataFrame من قاموس من القوائم

data = { 'الاسم': ['علي', 'سارة', 'أحمد', 'ليلى'], 'العمر': [25, 30, 35, 28], 'المدينة': ['الرياض', 'جدة', 'الدمام', 'القاهرة'] } df = pd.DataFrame(data) print("DataFrame:\n", df)

# إنشاء DataFrame من قائمة من القواميس

data_list = [ {'الاسم': 'خالد', 'العمر': 22, 'المدينة': 'مكة'}, {'الاسم': 'نورة', 'العمر': 24, 'المدينة': 'المدينة'} ] df_new = pd.DataFrame(data_list) print("\nDataFrame جديد:\n", df_new)
```

- **قراءة البيانات من مصادر مختلفة:** Pandas تُسهل قراءة البيانات من مجموعة متنوعة من المصادر، مثل ملفات Excel ،CSV، قواعد البيانات، JSON، وغيرها.

python```

مثال (يتطلب وجود ملف people.csv)

df_csv = pd.read_csv('people.csv')

print(df_csv)

```

- **الوصول إلى البيانات (Indexing and Selection):**
  - **الوصول إلى عمود:** باستخدام اسم العمود. `python print("\nعمود الاسم:\", n",` `df['الاسم']) print("\nعمود العمر:\", n", df.العمر) # طريقة أخرى للوصول`
  - **الوصول إلى صفوف:** باستخدام `loc` (للوصول بالاسم/التسمية) أو `iloc` (للوصول بالفهرس الرقمي). `python print("\nالصف الأول (باستخدام iloc):\", n",` `df.iloc[0]) print("\nالصفوف من 0 إلى 1 (باستخدام iloc):\", n",` `df.iloc[0:2])`
  - **التصفية الشرطية:** اختيار الصفوف بناءً على شرط معين. ```python filtered_df = df[df['العمر'] > 28] print("\nالبيانات المصفاة (العمر > 28):\", n", filtered_df)

## شروط متعددة

filtered_df_multi = df[(df['العمر'] > 25) & (df['المدينة'] == 'الرياض')] print("\nالبيانات المصفاة (العمر > 25 والمدينة الرياض):\", n", filtered_df_multi) ```

- **معالجة البيانات المفقودة:** Pandas توفر دوال قوية للتعامل مع القيم المفقودة (NaN - Not a Number).

```python df_missing = pd.DataFrame({ 'A': [1, 2, np.nan, 4], 'B': [5, np.nan, np.nan, 8], 'C': [9, 10, 11, 12] }) print("\nDataFrame مع قيم مفقودة:\", n", df_missing)

# إسقاط الصفوف التي تحتوي على قيم مفقودة

print("\nبعد إسقاط الصفوف ذات القيم المفقودة:\n", df_missing.dropna())

# ملء القيم المفقودة بقيمة معينة (مثلاً المتوسط)

print("\nبعد ملء القيم المفقودة بالمتوسط:\n", df_missing.fillna(df_missing.mean())) ```

- **العمليات الإحصائية والتلخيصية:** يمكنك بسهولة حساب الإحصائيات الوصفية، مثل المتوسط، الوسيط، الانحراف المعياري، إلخ.

```
python print("\nإحصائيات وصفية لعمود العمر:\n",
df['العمر'].describe()) print("متوسط العمر:", df['العمر'].mean())
print("أقصى عمر:", df['العمر'].max())
```

- **التجميع (Group By):** تُستخدم لتجميع البيانات بناءً على عمود واحد أو أكثر، ثم تطبيق دالة تجميعية (مثل `count`, `mean`, `sum`) على كل مجموعة.

```python data_sales = { 'المنتج': ['A', 'B', 'A', 'C', 'B', 'A'], 'الكمية': [10, 15, 12, 8, 20, 11], 'السعر': [100, 50, 110, 200, 60, 90] } df_sales = pd.DataFrame(data_sales) print("\nبيانات المبيعات:\n", df_sales)

# تجميع حسب المنتج وحساب إجمالي الكمية

total_quantity_by_product = df_sales.groupby('المنتج')['الكمية'].sum() print("\nإجمالي الكمية لكل منتج:\n", total_quantity_by_product) ```
```

3. Matplotlib - تفصيل إضافي

Matplotlib هي مكتبة شاملة لإنشاء تصورات ثابتة ومتحركة وتفاعلية في بايثون. تُعد الأداة الأكثر شيوعًا لإنشاء الرسوم البيانية في بايثون، وتوفر تحكمًا دقيقًا في كل جانب من جوانب الرسم البياني.

الميزات الرئيسية لـ Matplotlib:

- **أنواع الرسوم البيانية:** تدعم مجموعة واسعة من الرسوم البيانية، بما في ذلك:
  - **الرسوم البيانية الخطية (Line Plots):** لعرض الاتجاهات بمرور الوقت.
  - **الرسوم البيانية المبعثرة (Scatter Plots):** لإظهار العلاقة بين متغيرين.
  - **الرسوم البيانية الشريطية (Bar Plots):** لمقارنة الفئات.
  - **المدرجات التكرارية (Histograms):** لتوزيع البيانات.
  - **الرسوم البيانية الدائرية (Pie Charts):** لإظهار النسب المئوية.
  - **المربعات الصندوقية (Box Plots):** لتوزيع البيانات وإظهار القيم الشاذة.
- **البنية الأساسية:** تتكون Matplotlib من عدة طبقات، ولكن الواجهة الأكثر استخدامًا هي `pyplot`، والتي توفر واجهة تشبه MATLAB.

python import matplotlib.pyplot as plt import numpy as np```

1. إنشاء البيانات

x = np.linspace(0, 10, 100) # 100 نقطة بين 0 و 10 y = np.sin(x)

2. إنشاء الرسم البياني (Figure) والمحاور (Axes)

fig, ax = plt.subplots() # طريقة موصى بها لإنشاء الرسم البياني والمحاور

3. رسم البيانات على المحاور

ax.plot(x, y, label='sin(x)', color='blue', linestyle='--')

4. إضافة التسميات والعناوين والوسائل الإيضاحية

ax.set_xlabel("قيمة X") ax.set_ylabel("قيمة Y") ax.set_title("رسم دالة الجيب") ax.legend() ax.grid(True)

5. عرض الرسم البياني

plt.show() ```

- **أمثلة على أنواع الرسوم البيانية:**
    - **رسم بياني مبعثر (Scatter Plot):** ```python # بيانات عشوائية np.random.seed(42) x_data = np.random.rand(50) y_data = np.random.rand(50) colors = np.random.rand(50) sizes = 1000 * np.random.rand(50)

      plt.scatter(x_data, y_data, c=colors, s=sizes, alpha=0.7, cmap='viridis') plt.xlabel("المتغير 1") plt.ylabel("المتغير 2") plt.title("رسم بياني مبعثر") plt.colorbar(label="اللون") plt.show() ```
    - **رسم بياني شريطي (Bar Plot):** ```python categories = ['الفئة أ', 'الفئة ب', 'الفئة ج', 'الفئة د'] values = [23, 45, 56, 12]

      plt.bar(categories, values, color=['skyblue', 'lightcoral', 'lightgreen', 'gold']) plt.xlabel("الفئات") plt.ylabel("القيم") plt.title("مقارنة القيم بين الفئات") plt.show() ```
    - **مدرج تكراري (Histogram):** ```python data = np.random.randn(1000) # بيانات من توزيع طبيعي

("القيم")plt.hist(data, bins=30, edgecolor='black', alpha=0.7) plt.xlabel ("التكرار")plt.ylabel ("توزيع البيانات")plt.title ()plt.show ```

- **رسم بياني دائري (Pie Chart):** ```python labels = ['تفاح', 'برتقال', 'موز', 'عنب'] sizes = [15, 30, 45, 10] explode = (0, 0.1, 0, 0) # فصل شريحة "برتقال"

plt.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True, startangle=90) plt.axis('equal') # يضمن أن الدائرة مرسومة كدائرة ("توزيع الفاكهة")plt.title ()plt.show ```

ملاحظات هامة:

- **التثبيت:** إذا لم تكن هذه المكتبات مثبتة لديك، يمكنك تثبيتها باستخدام `bash pip` : `pip install numpy pandas matplotlib`
- **العرض:** لكي تعمل أمثلة Matplotlib وتُعرض الرسوم البيانية، يجب تشغيل الكود في بيئة تدعم عرض الرسوم البيانية، مثل:
    - **Jupyter Notebook / JupyterLab:** بيئة تفاعلية ممتازة لعلم البيانات.
    - **IDE مثل VS Code أو PyCharm:** غالبًا ما تحتوي على دعم مدمج لعرض الرسوم البيانية.
    - **تشغيل ملف .py من الطرفية:** سيتم فتح نافذة منفصلة لعرض الرسم البياني. يجب أن يكون لديك بيئة رسومية مثبتة.

هذه المكتبات الثلاث هي مجرد غيض من فيض ما تقدمه بايثون في مجال تحليل البيانات والتصور. إتقانها سيفتح لك أبوابًا واسعة في العديد من المجالات المتقدمة.

توسيع الفصل العاشر: مشاريع تطبيقية بسيطة

بعد أن استعرضنا أساسيات بايثون، وهياكل البيانات، والدوال، والبرمجة الكائنية التوجه، والتعامل مع الملفات، وحتى مقدمة للمكتبات الشائعة، حان الوقت لتطبيق كل هذه المعرفة في بناء مشاريع عملية. بناء المشاريع هو أفضل طريقة لترسيخ المفاهيم، وتطوير مهارات حل المشكلات، والتعامل مع التحديات الحقيقية التي تواجه المبرمجين. في هذا الفصل، سنقوم ببناء ثلاثة مشاريع بسيطة ولكنها مفيدة، مع شرح تفصيلي لكل خطوة.

1. مشروع آلة حاسبة بسيطة - تفصيل إضافي

هدف هذا المشروع هو بناء آلة حاسبة نصية (console-based) يمكنها إجراء العمليات الحسابية الأساسية: الجمع، الطرح، الضرب، والقسمة. سنستخدم الدوال لتنظيم الكود، والجمل الشرطية لاختيار العملية، وحلقات التكرار للسماح للمستخدم بإجراء عمليات متعددة.

الخطوات الأساسية:

1. **تعريف الدوال للعمليات الحسابية:** كل عملية (جمع، طرح، ضرب، قسمة) ستكون دالة منفصلة.
2. **عرض قائمة بالعمليات:** إظهار خيارات للمستخدم لاختيار العملية المطلوبة.
3. **الحصول على مدخلات المستخدم:** طلب الأرقام والعملية من المستخدم.
4. **تنفيذ العملية:** استدعاء الدالة المناسبة بناءً على اختيار المستخدم.
5. **معالجة الأخطاء:** التعامل مع المدخلات غير الصالحة (مثل إدخال نص بدلاً من رقم) والقسمة على صفر.
6. **السماح بعمليات متعددة:** تمكين المستخدم من إجراء عدة عمليات دون إعادة تشغيل البرنامج.

```python
def add(x, y):
    """تُرجع مجموع رقمين."""
    return x + y

def subtract(x, y):
    """تُرجع الفرق بين رقمين."""
    return x - y

def multiply(x, y):
    """تُرجع حاصل ضرب رقمين."""
    return x * y

def divide(x, y):
    """تُرجع حاصل قسمة رقمين. تُعالج حالة القسمة على صفر"""
    if y == 0:
        return "خطأ: لا يمكن القسمة على صفر!"
    return x / y

def power(x, y):
    """تُرجع x مرفوعة للقوة y."""
    return x ** y

print("-----------------------------------")
print("أهلاً بك في الآلة الحاسبة البسيطة")
print("-----------------------------------")
print("اختر العملية:")
print("1. جمع (+)")
print("2. طرح (-)")
print("3. ضرب (*)")
print("4. قسمة (/)")
print("5. أس (**)")
print("-----------------------------------")

while True:
    choice = input("أدخل اختيارك (1/2/3/4/5): ")

    if choice in ("1", "2", "3", "4", "5"):
        try:
            num1 = float(input("أدخل الرقم الأول: "))
            num2 = float(input("أدخل الرقم الثاني: "))
        except ValueError:
            print("\n*** خطأ: مدخل غير صالح. يرجى إدخال أرقام فقط. ***\n")
            continue # الانتقال إلى التكرار التالي للحلقة

        if choice == "1":
            print(f"{num1} + {num2} = {add(num1, num2)}")
        elif choice == "2":
            print(f"{num1} - {num2} = {subtract(num1, num2)}")
        elif choice == "3":
            print(f"{num1} * {num2} = {multiply(num1, num2)}")
```

```
        elif choice == "4":
            result = divide(num1, num2)
            print(f"{num1} / {num2} = {result}")
        elif choice == "5":
            print(f"{num1} ** {num2} = {power(num1, num2)}")

        print("---------------------------------")
        next_calculation = input("هل تريد إجراء عملية أخرى؟ (نعم/لا) ")
        if next_calculation.lower() == "لا":
            print("شكراً لاستخدام الآلة الحاسبة. إلى اللقاء!")
            break # إنهاء الحلقة والخروج من البرنامج
        print("---------------------------------")
    else:
        print("\n*** خطأ: اختيار غير صالح. يرجى إدخال 1 أو 2 أو 3 أو 4 أو 5
***\n")
```

تحليل الكود:

- **الدوال:** كل عملية حسابية تم تغليفها في دالة خاصة بها، مما يجعل الكود منظمًا وسهل القراءة وإعادة الاستخدام. دالة `divide` تحتوي على منطق خاص لمعالجة القسمة على صفر.
- **حلقة `while True`:** تسمح للبرنامج بالاستمرار في العمل حتى يقرر المستخدم الخروج.
- **`input()`:** تُستخدم للحصول على مدخلات من المستخدم.
- **`float()`:** تُستخدم لتحويل المدخلات النصية إلى أرقام عشرية. هذا يسمح بالتعامل مع الأعداد الصحيحة والعشرية.
- **`try-except ValueError`:** تُستخدم لمعالجة الأخطاء إذا أدخل المستخدم نصًا غير رقمي. `continue` تُعيد الحلقة إلى بدايتها.
- **الجمل الشرطية (`if-elif-else`):** تُستخدم لتحديد العملية التي اختارها المستخدم واستدعاء الدالة المناسبة.
- **`next_calculation.lower() == "لا"`:** تُستخدم لتحويل مدخل المستخدم إلى أحرف صغيرة للمقارنة، مما يجعل البرنامج مرنًا (يقبل "لا" أو "لا").
- **`break`:** تُستخدم لإنهاء حلقة `while` عندما يقرر المستخدم عدم إجراء عمليات أخرى.

2. مشروع لعبة تخمين الأرقام - تفصيل إضافي

في هذه اللعبة، سيقوم الكمبيوتر بتوليد رقم عشوائي ضمن نطاق محدد، وعلى المستخدم تخمين هذا الرقم. سيقدم الكمبيوتر تلميحات (أكبر أو أصغر) لمساعدة المستخدم، وسيكون هناك عدد محدود من المحاولات.

الخطوات الأساسية:

1. **استيراد وحدة `random`** : لتوليد الأرقام العشوائية.
2. **تحديد نطاق الرقم السري وعدد المحاولات:** لجعل اللعبة قابلة للتخصيص.
3. **توليد الرقم السري:** باستخدام `random.randint()` .
4. **حلقة اللعبة الرئيسية:** تستمر طالما أن المستخدم لديه محاولات ولم يخمن الرقم.
5. **الحصول على تخمين المستخدم:** طلب رقم من المستخدم.
6. **معالجة الأخطاء:** التأكد من أن المدخل رقم صحيح.
7. **تقديم التلميحات:** إخبار المستخدم ما إذا كان تخمينه أكبر أو أصغر من الرقم السري.
8. **التحقق من الفوز أو الخسارة:** إنهاء اللعبة عند التخمين الصحيح أو نفاد المحاولات.

```python
import random

def guess_the_number_game():
    print("----------------------------------")
    print("أهلاً بك في لعبة تخمين الأرقام!")
    print("----------------------------------")

    # تحديد نطاق الرقم السري
    lower_bound = 1
    upper_bound = 100
    secret_number = random.randint(lower_bound, upper_bound) # يولد رقماً عشوائياً بين 1 و 100

    max_attempts = 7 # عدد المحاولات المسموح بها
    attempts = 0

    print(f"لقد اخترت رقماً سرياً بين {lower_bound} و {upper_bound}. لديك {max_attempts} محاولات لتخمينه.")

    while attempts < max_attempts:
        try:
            guess = int(input(f"المحاولة رقم {attempts + 1}/{max_attempts}: أدخل تخمينك: "))
            attempts += 1

            if guess < lower_bound or guess > upper_bound:
                print(f"*** يرجى التخمين ضمن النطاق ({lower_bound}-{upper_bound}). ***")
                continue # لا نحسب هذه المحاولة إذا كانت خارج النطاق

            if guess < secret_number:
                print("الرقم السري أكبر! حاول مرة أخرى.")
            elif guess > secret_number:
                print("الرقم السري أصغر! حاول مرة أخرى.")
            else:
                print("----------------------------------")
                print(f"تهانينا! لقد خمنت الرقم السري ({secret_number}) في {attempts} محاولات.")
                print("----------------------------------")
                return # إنهاء الدالة والخروج من اللعبة
        except ValueError:
            print("\n*** خطأ: مدخل غير صالح. يرجى إدخال رقم صحيح ***\n")
            # لا نزيد عدد المحاولات هنا لأن المدخل لم يكن رقماً

    # إذا وصلت الحلقة إلى هنا، فهذا يعني أن المحاولات قد نفدت
    print("----------------------------------")
    print(f"لقد نفدت محاولاتك! الرقم السري كان {secret_number}.")
    print("حظاً أوفر في المرة القادمة!")
    print("----------------------------------")
```

```
# استدعاء اللعبة لبدئها
guess_the_number_game()
```

تحليل الكود:

- `import random`: لاستخدام دالة `random.randint()` لتوليد الرقم العشوائي.
- `max_attempts`, `upper_bound`, `lower_bound`: متغيرات لتخصيص اللعبة.
- **حلقة** `while attempts < max_attempts`: تستمر اللعبة طالما أن عدد المحاولات لم يتجاوز الحد الأقصى.
- `try-except ValueError`: لمعالجة الأخطاء إذا أدخل المستخدم شيئًا غير رقم.
- **التحقق من النطاق:** يضمن أن التخمين يقع ضمن النطاق المحدد. إذا كان خارج النطاق، لا تُحسب المحاولة.
- **التلميحات:** تُقدم للمستخدم إرشادات حول ما إذا كان الرقم السري أكبر أو أصغر.
- `return`: تُستخدم لإنهاء الدالة فورًا عند الفوز.
- **رسالة الخسارة:** تُعرض إذا نفدت المحاولات دون تخمين الرقم الصحيح.

3. مشروع إدارة قائمة المهام (To-Do List) - تفصيل إضافي

هذا المشروع سيمكن المستخدم من إدارة قائمة مهامه اليومية. سيتضمن البرنامج وظائف لإضافة مهام جديدة، عرض جميع المهام، وضع علامة على مهمة كمكتملة، وحذف مهمة. سنستخدم قائمة بايثون لتخزين المهام، وقواميس لتمثيل كل مهمة (اسم المهمة، حالة الاكتمال).

الخطوات الأساسية:

1. **هيكل البيانات:** استخدام قائمة من القواميس لتخزين المهام.
2. **الدوال للعمليات:** كل عملية (إضافة، عرض، إكمال، حذف) ستكون دالة منفصلة.
3. **القائمة الرئيسية:** عرض خيارات للمستخدم في حلقة تكرارية.
4. **الحصول على مدخلات المستخدم:** طلب اختيار العملية وأي تفاصيل إضافية (مثل اسم المهمة أو رقمها).
5. **معالجة الأخطاء:** التأكد من أن المدخلات صحيحة (مثل رقم المهمة).

```python
tasks = [] # قائمة عالمية لتخزين المهام

def add_task(task_name):
    """تُضيف مهمة جديدة إلى القائمة."""
    tasks.append({"name": task_name, "completed": False})
    print(f"\n>>> تم إضافة المهمة: \'{task_name}\' بنجاح.\n")

def view_tasks():
    """تعرض جميع المهام في القائمة مع حالتها."""
    if not tasks:
        print("\n>>> لا توجد مهام حالياً في القائمة. ابدأ بإضافة بعض المهام!\n")
        return

    print("\n--- قائمة المهام الحالية ---")
    for i, task in enumerate(tasks):
        status = "[مكتمل ✔]" if task["completed"] else "[غير مكتمل ⏳]"
        print(f"{i + 1}. {task["name"]} {status}")
    print("---------------------------\n")

def complete_task(task_index):
    """تضع علامة على مهمة معينة كمكتملة."""
    if 0 <= task_index < len(tasks):
        if not tasks[task_index]["completed"]:
            tasks[task_index]["completed"] = True
            print(f"\n>>> تم وضع علامة على المهمة \'{tasks[task_index]
["name"]}\' كمكتملة.\n")
        else:
            print(f"\n>>> المهمة \'{tasks[task_index]["name"]}\' مكتملة
بالفعل.\n")
    else:
        print("\n*** خطأ: رقم مهمة غير صالح. يرجى التحقق من القائمة. ***\n")

def delete_task(task_index):
    """تحذف مهمة معينة من القائمة."""
    if 0 <= task_index < len(tasks):
        removed_task = tasks.pop(task_index)
        print(f"\n>>> تم حذف المهمة: \'{removed_task["name"]}\' بنجاح.\n")
    else:
        print("\n*** خطأ: رقم مهمة غير صالح. يرجى التحقق من القائمة. ***\n")

# الحلقة الرئيسية للبرنامج
while True:
    print("--- برنامج إدارة المهام ---")
    print("1. إضافة مهمة جديدة")
    print("2. عرض جميع المهام")
    print("3. إكمال مهمة")
    print("4. حذف مهمة")
    print("5. خروج")
    print("---------------------------")
```

```python
    choice = input("أدخل اختيارك (5-1): ")

    if choice == "1":
        task_name = input("أدخل اسم المهمة الجديدة: ")
        add_task(task_name)
    elif choice == "2":
        view_tasks()
    elif choice == "3":
        view_tasks()
        if tasks: # التحقق مما إذا كانت هناك مهام لإكمالها
            try:
                task_num = int(input("أدخل رقم المهمة التي تريد إكمالها: "))
                complete_task(task_num - 1) # -1 لأن القوائم تبدأ من 0
            except ValueError:
                print("\n*** خطأ: مدخل غير صالح. يرجى إدخال رقم صحيح للمهمة.
***\n")
    elif choice == "4":
        view_tasks()
        if tasks: # التحقق مما إذا كانت هناك مهام لحذفها
            try:
                task_num = int(input("أدخل رقم المهمة التي تريد حذفها: "))
                delete_task(task_num - 1) # -1 لأن القوائم تبدأ من 0
            except ValueError:
                print("\n*** خطأ: مدخل غير صالح. يرجى إدخال رقم صحيح للمهمة.
***\n")
    elif choice == "5":
        print("\nشكراً لاستخدامك برنامج إدارة المهام. إلى اللقاء!\n")
        break # إنهاء الحلقة والخروج من البرنامج
    else:
        print("\n*** خطأ: اختيار غير صالح. يرجى إدخال رقم بين 1 و 5. ***\n")
```

تحليل الكود:

- `tasks = []`**:** قائمة فارغة تُستخدم لتخزين جميع المهام. كل مهمة هي قاموس يحتوي على `"name"` و `"completed"`.
- **الدوال:** كل وظيفة (إضافة، عرض، إكمال، حذف) لها دالة خاصة بها، مما يجعل الكود معياريًا وسهل الفهم.
- `enumerate(tasks)`**:** تُستخدم في `view_tasks()` للحصول على الفهرس والقيمة لكل مهمة، مما يسهل عرض المهام بأرقام متسلسلة للمستخدم.
- `if not tasks:`**:** تُستخدم للتحقق مما إذا كانت القائمة فارغة قبل محاولة عرض المهام أو إكمالها/حذفها.
- `task_num - 1`**:** يتم طرح 1 من رقم المهمة الذي يدخله المستخدم لأن المستخدمين عادةً ما يبدأون العد من 1، بينما تبدأ فهارس القوائم في بايثون من 0.

- **معالجة الأخطاء:** تُستخدم `try-except ValueError` للتعامل مع المدخلات غير الرقمية عند طلب رقم المهمة.
- **حلقة `while True`:** تُبقي البرنامج قيد التشغيل حتى يختار المستخدم الخروج.

هذه المشاريع الثلاثة تُظهر كيفية تطبيق المفاهيم التي تعلمتها في هذا الكتاب لبناء تطبيقات عملية. لا تتردد في تعديلها، وتوسيعها، وإضافة ميزات جديدة إليها. الممارسة هي مفتاح إتقان البرمجة!